



low level

TEX

lowlevel

Contents

2	Conditionals	7
2.1	Preamble	8
2.2	\TeX primitives	13
2.3	$\varepsilon\text{-}\text{\TeX}$ primitives	20
2.4	\LuaTeX primitives	24
2.5	\LuaMetaTeX primitives	28
2.6	For the brave	40
2.7	Relaxing	42
3	Boxes	46
3.1	Introduction	47
3.2	Boxes	47
3.3	\TeX primitives	48
3.4	$\varepsilon\text{-}\text{\TeX}$ primitives	51
3.5	\LuaTeX primitives	51
3.6	\LuaMetaTeX primitives	52
3.7	Splitting	60
4	Expansion	66
4.1	Preamble	67
4.2	\TeX primitives	67
4.3	$\varepsilon\text{-}\text{\TeX}$ primitives	72
4.4	\LuaTeX primitives	74
4.5	\LuaMetaTeX primitives	75
4.6	Dirty tricks	85
5	Registers	88
5.1	Preamble	89
5.2	\TeX primitives	89
5.3	$\varepsilon\text{-}\text{\TeX}$ primitives	92
5.4	\LuaTeX primitives	92
5.5	\LuaMetaTeX primitives	93
5.6	Units	94
6	Macros	100
6.1	Preamble	101
6.2	Definitions	101
6.3	Runaway arguments	111

6.4	Introspection	112
6.5	nesting	113
6.6	Prefixes	116
6.7	Arguments	118
6.8	Constants	119
6.9	Passing parameters	120
6.10	Nesting	124
6.11	Duplicate hashes	125
7	Grouping	132
7.1	Introduction	133
7.2	Kinds of grouping	135
7.3	Hooks	137
7.4	Local versus global	138
7.5	Files	141
8	Security	142
8.1	Preamble	143
8.2	Flags	143
8.3	Complications	146
8.4	Introspection	147
9	Characters	148
9.1	Introduction	149
9.2	History	149
9.3	The heritage	150
9.4	The LMTX approach	151
9.5	spaces	154
10	Scope	156
10.1	Introduction	157
10.2	Registers	157
10.3	Allocation	159
10.4	Files	162
11	Paragraphs	164
11.1	Introduction	165
11.2	Paragraphs	165
11.3	Properties	168
11.4	Wrapping up	171
11.5	Hanging	171
11.6	Shapes	171

11.7	Modes	190
11.8	Leaders	191
11.9	Prevdepth	197
11.10	Normalization	199
11.11	Dirty tricks	199
11.12	Penalties	199
11.13	Par passes	200
12	Alignments	203
12.1	Introduction	204
12.2	Between the lines	206
12.3	Pre-, inter- and post-tab skips	208
12.4	Cell widths	211
12.5	Plugins	212
12.6	Pitfalls and tricks	215
12.7	Rows	218
12.8	Templates	220
12.9	Pitfalls	221
12.10	Remark	223
13	Marks	224
13.1	Introduction	225
13.2	The basics	225
13.3	Migration	226
13.4	Tracing	229
13.5	High level commands	230
13.6	Pitfalls	232
14	Inserts	233
14.1	Introduction	234
14.2	The page builder	234
14.3	Inserts	236
14.4	Storing	237
14.5	Synchronizing	237
14.6	Migration	237
14.7	Callbacks	237
15	Localboxes	238
15.1	Introduction	239
15.2	The basics	239
15.3	The interface	242
15.4	The helpers	247

16 Loops	249
16.1 Introduction	250
16.2 Primitives	250
16.3 Wrappers	254
16.4 About quitting	256
16.5 Simple repeaters	256
16.6 Endless loops	256
16.7 Loop variables	256
17 Tokens	259
17.1 Introduction	260
17.2 What are tokens	260
17.3 Some implementation details	263
17.4 Other data management	263
17.5 Macros	264
17.6 Looking at tokens	264
18 Buffers	271
18.1 Preamble	272
18.2 Encoding	272
18.3 Performance	273
18.4 Files	274
18.5 Macros	274
18.6 Token lists	275
18.7 Buffers	275
18.8 Setups	278
18.9 xml	279
18.10 Lua	279
18.11 Protection	279
19 Accuracy	281
19.1 Introduction	282
19.2 Posits	283
19.3 MetaPost	286
19.4 Lua	287
20 Balancing	288
20.1 Introduction	289
20.2 Intercepting the MVL	289
20.3 Balancing	292
20.4 Forcing breaks	297

20.5	Marks	298
20.6	Inserts	298
20.7	Discardables	302
20.8	Passes	304
20.9	Passes	305
21	Lines	306
21.1	Introduction	307
21.2	Warning	308
21.3	Constructing paragraphs	308
21.4	Subpasses	312
21.5	Definitions	316
21.6	Tracing	318
21.7	Criterion	321
21.8	Examples	321
21.9	Pages	322
21.10	Profiles	325
22	Debugging	327
22.1	Introduction	328
22.2	Token lists	328
22.3	Node lists	331
22.4	Visual debugging	332
22.5	Math	333
22.6	Fonts	335
22.7	Overflow	336
22.8	Side floats	337
22.9	Struts	338
22.10	Features	338
22.11	Profiling	339
22.12	Par builder	340
22.13	More	341
23	Pages	342
23.1	Introduction	343
23.2	Rows becoming columns	343

Colofon

Author	Hans Hagen
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com contextgarden.net ntg-context@ntg.nl

2 Conditionals

low level

TEX

conditionals

Contents

2.1	Preamble	8
2.2	T _E X primitives	13
2.3	ε -T _E X primitives	20
2.4	LuaT _E X primitives	24
2.5	LuaMetaT _E X primitives	28
2.6	For the brave	40
2.7	Relaxing	42

2.1 Preamble

2.1.1 Introduction

You seldom need the low level conditionals because there are quite some so called support macros available in ConT_EXt. For instance, when you want to compare two values (or more accurate: sequences of tokens), you can do this:

```
\doifelse {foo} {bar} {
  the same
} {
  different
}
```

But if you look in the ConT_EXt code, you will see that often we use primitives that start with `\if` in low level macros. There are good reasons for this. First of all, it looks familiar when you also code in other languages. Another reason is performance but that is only true in cases where the snippet of code is expanded very often, because T_EX is already pretty fast. Using low level T_EX can also be more verbose, which is not always nice in a document source. But, the most important reason (for me) is the layout of the code. I often let the look and feel of code determine the kind of coding. This also relates to the syntax highlighting that I am using, which is consistent for T_EX, MetaPost, Lua, etc. and evolved over decades. If code looks bad, it probably is bad. Of course this doesn't mean all my code looks good; you're warned. In general we can say that I often use `\if...` when coding core macros, and `\doifelse...` macros in (document) styles and modules.

In the sections below I will discuss the low level conditions in T_EX. For the often more convenient ConT_EXt wrappers you can consult the source of the system and support modules, the wiki and/or manuals.

Some of the primitives shown here are only available in LuaT_EX, and some only in Lua-MetaT_EX. We could do without them for decades but they were added to these engines because of convenience and, more important, because then made for nicer code. Of course there's also the fun aspect. This manual is not an invitation to use these very low level primitives in your document source. The ones that probably make most sense are `\ifnum`, `\ifdim` and `\ifcase`. The others are often wrapped into support macros that are more convenient.

2.1.2 Number and dimensions

Numbers and dimensions are basic data types in T_EX. When you enter one, a number is just that but a dimension gets a unit. Compare:

```
1234
1234pt
```

If you also use MetaPost, you need to be aware of the fact that in that language there are not really dimensions. The `post` part of the name implies that eventually a number becomes a PostScript unit which represents a base point (bp) in T_EX. When in MetaPost you entry `1234pt` you actually multiply 1234 by the variable `pt`. In T_EX on the other hand, a unit like `pt` is one of the keywords that gets parsed. Internally dimensions are also numbers and the unit (keyword) tells the scanner what multiplier to use. When that multiplier is one, we're talking of scaled points, with the unit `sp`.

```
\the\dimexpr 12.34pt \relax
\the\dimexpr 12.34sp \relax
\the\dimexpr 12.99sp \relax
\the\dimexpr 1234sp  \relax
\the\numexpr 1234    \relax
```

```
12.34pt
0.00018pt
0.00018pt
0.01883pt
1234
```

When we serialize a dimension it always shows the dimension in points, unless we serialize it as number.

```
\scratchdimen1234sp
\number\scratchdimen
\the\scratchdimen
```

1234
0.01883pt

When a number is scanned, the first thing that is taken care of is the sign. In many cases, when T_EX scans for something specific it will ignore spaces. It will happily accept multiple signs:

```
\number +123
\number +++123
\number + + + 123
\number +--++123
\number --123
\number ---123
```

123
123
123
123
123
-123

Watch how the negation accumulates. The scanner can handle decimal, hexadecimal and octal numbers:

```
\number -123
\number -"123
\number -'123
```

-123
-291
-83

A dimension is scanned like a number but this time the scanner checks for upto three parts: an either or not signed number, a period and a fraction. Here no number means zero, so the next is valid:

```
\the\dimexpr . pt \relax
\the\dimexpr 1. pt \relax
\the\dimexpr .1pt \relax
\the\dimexpr 1.1pt \relax
```

0.0pt
1.0pt

0.1pt

1.1pt

Again we can use hexadecimal and octal numbers but when these are entered, there can be no fractional part.

```
\the\dimexpr 16 pt \relax
\the\dimexpr "10 pt \relax
\the\dimexpr '20 pt \relax
```

16.0pt

16.0pt

16.0pt

The reason for discussing numbers and dimensions here is that there are cases where when \TeX expects a number it will also accept a dimension. It is good to know that for instance a macro defined with `\chardef` or `\mathchardef` also is treated as a number. Even normal characters can be numbers, when prefixed by a ``` (backtick).

The maximum number in \TeX is 2147483647 so we can do this:

```
\scratchcounter2147483647
```

but not this

```
\scratchcounter2147483648
```

as it will trigger an error. A dimension can be positive and negative so there we can do at most:

```
\scratchdimen 1073741823sp
```

```
\scratchdimen1073741823sp
```

```
\number\scratchdimen
```

```
\the\scratchdimen
```

```
\scratchdimen16383.99998pt
```

```
\number\scratchdimen
```

```
\the\scratchdimen
```

1073741823

16383.99998pt

1073741823

16383.99998pt

We can also do this:

```
\scratchdimen16383.99999pt
\number\scratchdimen
\the\scratchdimen

1073741823
16383.99998pt
```

but the next one will fail:

```
\scratchdimen16383.9999999pt
```

Just keep in mind that T_EX scans both parts as number so the error comes from checking if those numbers combine well.

```
\ifdim 16383.99999 pt = 16383.99998 pt the same \else different \fi
\ifdim 16383.999979 pt = 16383.999980 pt the same \else different \fi
\ifdim 16383.999987 pt = 16383.999991 pt the same \else different \fi
```

Watch the difference in dividing, the / rounds, while the : truncates.

the same
the same
the same

You need to be aware of border cases, although in practice they never really are a problem:

```
\ifdim \dimexpr16383.99997 pt/2\relax = \dimexpr 16383.99998 pt/2\relax
the same \else different
\fi
\ifdim \dimexpr16383.99997 pt:2\relax = \dimexpr 16383.99998 pt:2\relax
the same \else different
\fi
```

different
the same

```
\ifdim \dimexpr1.99997 pt/2\relax = \dimexpr 1.99998 pt/2\relax
the same \else different
\fi
\ifdim \dimexpr1.99997 pt:2\relax = \dimexpr 1.99998 pt:2\relax
the same \else different
```

`\fi`

different
the same

```
\ifdim \dimexpr1.999999 pt/2\relax = \dimexpr 1.999995 pt/2\relax
the same \else different
```

`\fi`

```
\ifdim \dimexpr1.999999 pt:2\relax = \dimexpr 1.999995 pt:2\relax
the same \else different
```

`\fi`

the same
the same

This last case demonstrates that at some point the digits get dropped (still assuming that the fraction is within the maximum permitted) so these numbers then are the same. Anyway, this is not different in other programming languages and just something you need to be aware of.

2.2 T_EX primitives

2.2.1 `\if`

I seldom use this one. Internally T_EX stores (and thinks) in terms of tokens. If you see for instance `\def` or `\dimen` or `\hbox` these all become tokens. But characters like A or @ also become tokens. In this test primitive all non-characters are considered to be the same. In the next examples this is demonstrated.

```
[\if AB yes\else nop\fi]
[\if AA yes\else nop\fi]
[\if CDyes\else nop\fi]
[\if CCyes\else nop\fi]
[\if\dimen\font yes\else nop\fi]
[\if\dimen\font yes\else nop\fi]
```

Watch how spaces after the two characters are kept: [nop] [yes] [nop] [yes] [yes] [yes]. This primitive looks at the next two tokens but when doing so it expands. Just look at the following:

```
\def\AA{AA}%
```

```
\def\AB{AB}%
[\if\AA yes\else nop\fi]
[\if\AB yes\else nop\fi]
```

We get: [yes] [nop].

2.2.2 \ifcat

In T_EX characters (in the input) get interpreted according to their so called catcodes. The most common are letters (alphabetic) and other (symbols) but for instance the backslash has the property that it starts a command, the dollar signs trigger math mode, while the curly braced deal with grouping. If for instance either or not the ampersand is special (for instance as column separator in tables) depends on the macro package.

```
[\ifcat AB yes\else nop\fi]
[\ifcat AA yes\else nop\fi]
[\ifcat CDyes\else nop\fi]
[\ifcat CCyes\else nop\fi]
[\ifcat Clyes\else nop\fi]
[\ifcat\dimen\font yes\else nop\fi]
[\ifcat\dimen\font yes\else nop\fi]
```

This time we also compare a letter with a number: [yes] [yes] [yes] [yes] [nop] [yes] [yes]. In that case the category codes differ (letter vs other) but in this test comparing the letters result in a match. This is a test that is used only once in ConT_EXt and even that occasion is dubious and will go away.

You can use \noexpand to prevent expansion:

```
\def\A{A}%
\let\B B%
\def\C{D}%
\let\D D%
[\ifcat\noexpand\A Ayes\else nop\fi]
[\ifcat\noexpand\B Byes\else nop\fi]
[\ifcat\noexpand\C Cyes\else nop\fi]
[\ifcat\noexpand\C Dyes\else nop\fi]
[\ifcat\noexpand\D Dyes\else nop\fi]
```

We get: [nop] [yes] [nop] [nop] [yes], so who still thinks that T_EX is easy to understand for a novice user?

2.2.3 `\ifnum`

This condition compares its argument with another one, separated by an `<`, `=` or `>` character.

```
\ifnum\scratchcounter<0
    less than
\else\ifnum\scratchcounter>0
    more than
\else
    equal to
\fi zero
```

This is one of these situations where a dimension can be used instead. In that case the dimension is in scaled points.

```
\ifnum\scratchdimen<0
    less than
\else\ifnum\scratchdimen>0
    more than
\else
    equal to
\fi zero
```

Of course this equal treatment of a dimension and number is only true when the dimension is a register or box property.

2.2.4 `\ifdim`

This condition compares one dimension with another one, separated by an `<`, `=` or `>` sign.

```
\ifdim\scratchdimen<0pt
    less than
\else\ifdim\scratchdimen>0pt
    more than
\else
    equal to
\fi zero
```

While when comparing numbers a dimension is a valid quantity but here you cannot mix them: something with a unit is expected.

2.2.5 `\ifodd`

This one can come in handy, although in ConT_EXt it is only used in checking for an odd of even page number.

```
\scratchdimen 3sp
\scratchcounter4
```

```
\ifodd\scratchdimen very \else not so \fi odd
\ifodd\scratchcounter very \else not so \fi odd
```

As with the previously discussed `\ifnum` you can use a dimension variable too, which is then interpreted as representing scaled points. Here we get:

```
very odd
not so odd
```

2.2.6 `\ifvmode`

This is a rather trivial check. It takes no arguments and just is true when we're in vertical mode. Here is an example:

```
\hbox{\ifvmode\else\par\fi\ifvmode v\else h\fi mode}
```

We're always in horizontal mode and issuing a `\par` inside a horizontal box doesn't change that, so we get: `hmode`.

2.2.7 `\ifhmode`

As with `\ifvmode` this one has no argument and just tells if we're in vertical mode.

```
\vbox {
  \noindent \ifhmode h\else v\fi mode
  \par
  \ifhmode h\else \noindent v\fi mode
}
```

You can use it for instance to trigger injection of code, or prevent that some content (or command) is done more than once:

```
hmode
vmode
```

2.2.8 `\ifmmode`

Math is something very \TeX so naturally you can check if you're in math mode. here is an example of using this test:

```
\def\enforcemath#1{\ifmmode#1\else$ #1 $\fi}
```

Of course in reality macros that do such things are more advanced than this one.

2.2.9 `\ifinner`

```
\def\ShowMode
  {\ifhmode      \ifinner inner \fi hmode
   \else\ifvmode \ifinner inner \fi vmode
   \else\ifmmode \ifinner inner \fi mmode
   \else        \ifinner inner \fi unset
   \fi\fi\fi}
```

```
\ShowMode \ShowMode
```

```
\vbox{\ShowMode}
```

```
\hbox{\ShowMode}
```

```
$\ShowMode$
```

```
$$\ShowMode$$
```

The first line has two tests, where the first one changes the mode to horizontal simply because a text has been typeset. Watch how display math is not inner.

```
vmode hmode
inner vmode
inner hmode
innermmode
innermmode
```

By the way, moving the `\ifinner` test outside the branches (to the top of the macro) won't work because once the word `inner` is typeset we're no longer in vertical mode, if we were at all.

2.2.10 `\ifvoid`

A box is one of the basic concepts in \TeX . In order to understand this primitive we present four cases:

```
\setbox0\hbox{}          \ifvoid0 void \else content \fi
\setbox0\hbox{123}        \ifvoid0 void \else content \fi
\setbox0\hbox{} \box0      \ifvoid0 void \else content \fi
\setbox0\hbox to 10pt{}    \ifvoid0 void \else content \fi
```

In the first case, we have a box which is empty but it's not void. It helps to know that internally an `hbox` is actually an object with a pointer to a linked list of nodes. So, the first two can be seen as:

```
hlist -> [nothing]
hlist -> 1 -> 2 -> 3 -> [nothing]
```

but in any case there is a `hlist`. The third case puts something in a `hlist` but then flushes it. Now we have not even the `hlist` any more; the box register has become void. The last case is a variant on the first. It is an empty box with a given width. The outcome of the four lines (with a box flushed in between) is:

```
content
content
```

```
void
content
```

So, when you want to test if a box is really empty, you need to test also its dimensions, which can be up to three tests, depending on your needs.

```
\setbox0\emptybox          \ifvoid0 void\else content\fi
\setbox0\emptybox          \wd0=10pt \ifvoid0 void\else content\fi
\setbox0\hbox to 10pt {}    \ifvoid0 void\else content\fi
\setbox0\hbox               {} \wd0=10pt \ifvoid0 void\else content\fi
```

Setting a dimension of a void (empty) box doesn't make it less void:

```
void
void
content
content
```

2.2.11 `\ifhbox`

This test takes a box number and gives true when it is an hbox.

2.2.12 `\ifvbox`

This test takes a box number and gives true when it is an vbox. Both a `\vbox` and `\vtop` are vboxes, the difference is in the height and depth and the baseline. In a `\vbox` the last line determines the baseline

`\vbox or \vtop`

`\vtop or \vbox`

And in a `\vtop` the first line takes control:

`\vbox or \vtop`

`\vtop or \vbox`

but, once wrapped, both internally are just vlists.

2.2.13 `\ifx`

This test is actually used a lot in `ConTEXt`: it compares two token(list)s:

```

\ifx a b Y\else N\fi
\ifx ab Y\else N\fi
\def\A {a}\def\B{b}\ifx \A\B Y\else N\fi
\def\A{aa}\def\B{a}\ifx \A\B Y\else N\fi
\def\A {a}\def\B{a}\ifx \A\B Y\else N\fi

```

Here the result is: “NNNNY”. It does not expand the content, if you want that you need to use an `\edef` to create two (temporary) macros that get compared, like in:

```

\edef\TempA{...}\edef\TempB{...}\ifx\TempA\TempB ...\else ... \fi

```

2.2.14 `\ifeof`

This test checks if a the pointer in a given input channel has reached its end. It is also true when the file is not present. The argument is a number which relates to the `\openin` primitive that is used to open files for reading.

2.2.15 `\iftrue`

It does what it says: always true.

2.2.16 `\iffalse`

It does what it says: always false.

2.2.17 `\ifcase`

The general layout of an `\ifcase` tests is as follows:

```
\ifcase<number>
    when zero
\or
    when one
\or
    when two
\or
    ...
\else
    when something else
\fi
```

As in other places a number is a sequence of signs followed by one or more digits

2.3 ε -T_EX primitives

2.3.1 `\ifdefined`

This primitive was introduced for checking the existence of a macro (or primitive) and with good reason. Say that you want to know if `\MyMacro` is defined? One way to do that is:

```
\ifx\MyMacro\undefined
    {\bf undefined indeed}
\fi
```

This results in: **undefined indeed**, but is this macro really undefined? When T_EX scans your source and sees the escape character (the forward slash) it will grab the next

characters and construct a control sequence from it. Then it finds out that there is nothing with that name and it will create a hash entry for a macro with that name but with no meaning. Because `\undefined` is also not defined, these two macros have the same meaning and therefore the `\ifx` is true. Imagine that you do this many times, with different macro names, then your hash can fill up. Also, when a user defined `\undefined` you're suddenly get a different outcome.

In order to catch the last problem there is the option to test directly:

```
\ifdefined\MyOtherMacro \else
  {\bf also undefined}
\fi
```

This (or course) results in: **also undefined**, but the macro is still sort of defined (with no meaning). The next section shows how to get around this.

2.3.2 `\ifcsname`

A macro is often defined using a ready made name, as in:

```
\def\OhYes{yes}
```

The name is made from characters with catcode letter which means that you cannot use for instance digits or underscores unless you also give these characters that catcode, which is not that handy in a document. You can however use `\csname` to define a control sequence with any character in the name, like:

```
\expandafter\def\csname Oh Yes : 1\endcsname{yes}
```

Later on you can get this one with `\csname`:

```
\csname Oh Yes : 1\endcsname
```

However, if you say:

```
\csname Oh Yes : 2\endcsname
```

you won't get some result, nor a message about an undefined control sequence, but the name triggers a define anyway, this time not with no meaning (undefined) but as equivalent to `\relax`, which is why

```
\expandafter\ifx\csname Oh Yes : 2\endcsname\relax
  {\bf relaxed indeed}
```

`\fi`

is the way to test its existence. As with the test in the previous section, this can deplete the hash when you do lots of such tests. The way out of this is:

```
\ifcsname Oh Yes : 2\endcsname \else
  {\bf unknown indeed}
\fi
```

This time there is no hash entry created and therefore there is not even an undefined control sequence.

In LuaTeX there is an option to return false in case of a messy expansion during this test, and in LuaMetaTeX that is default. This means that tests can be made quite robust as it is pretty safe to assume that names that make sense are constructed from regular characters and not boxes, font switches, etc.

2.3.3 `\iffontchar`

This test was also part of the ε -TeX extensions and it can be used to see if a font has a character.

```
\iffontchar\font`A
  {\em This font has an A!}
\fi
```

And, as expected, the outcome is: “*This font has an A!*”. The test takes two arguments, the first being a font identifier and the second a character number, so the next checks are all valid:

```
\iffontchar\font`A yes\else nop\fi\par
\iffontchar\nullfont`A yes\else nop\fi\par
\iffontchar\textfont0`A yes\else nop\fi\par
```

In the perspective of LuaMetaTeX I considered also supporting `\fontid` but it got a bit messy due to the fact that this primitive expands in a different way so this extension was rejected.

2.3.4 `\unless`

You can negate the results of a test by using the `\unless` prefix, so for instance you can replace:

```

\ifdim\scratchdimen=10pt
  \dosomething
\else\ifdim\scratchdimen<10pt
  \dosomething
\fi\fi

```

by:

```

\unless\ifdim\scratchdimen>10pt
  \dosomething
\fi

```

An `\unless` makes little sense when used with `\ifcase` but contrary to the other engines we don't error or it; we just give a warning. Some conditionals internally use a case so there we can actually provide a variant:

```

\ifcase 1 \relax zero \or one \or two \else else \fi = one \par
\ifcase 2 \relax zero \or one \or two \else else \fi = two \par

\unless\ifcase 1 \relax zero \or one \or two \else else \fi % warning
\unless\ifcase 2 \relax zero \or one \or two \else else \fi % warning

\ifchkdim1pt\or yes \else nop \fi = yes \par
\ifchkdim2 \or nop \else yes \fi = yes \par

\unless\ifchkdim1pt\or nop \else yes \fi = yes \par
\unless\ifchkdim2 \or yes \else nop \fi = yes \par

```

The `\ifchkdim`, `\ifchkdimension`, `\ifchknum`, `\ifchknumber` and `\ifparameter` are supported.

```

one = one
two = two
one two
yes = yes
yes = yes
yes = yes
yes = yes

```

2.4 Lua_T_EX primitives

2.4.1 `\ifincsname`

As it had no real practical usage it might get dropped in LuaMeta_T_EX, so it will not be discussed here.

2.4.2 `\ifprimitive`

As it had no real practical usage due to limitations, this one is not available in LuaMeta_T_EX so it will not be discussed here. If really needed you can use `\ifflags`.

2.4.3 `\ifabsnum`

This test is inherited from pdf_T_EX and behaves like `\ifnum` but first turns a negative number into a positive one.

2.4.4 `\ifabsdim`

This test is inherited from pdf_T_EX and behaves like `\ifdim` but first turns a negative dimension into a positive one.

2.4.5 `\ifcondition`

This is not really a test but in order to unstand that you need to know how _T_EX internally deals with tests.

```
\ifdimen\scratchdimen>10pt
  \ifdim\scratchdimen<20pt
    result a
  \else
    result b
  \fi
\else
  result c
\fi
```

When we end up in the branch of “result a” we need to skip two `\else` branches after we're done. The `\if..` commands increment a level while the `\fi` decrements a level.

The `\else` needs to be skipped here. In other cases the true branch needs to be skipped till we end up at the right `\else`. When doing this skipping, \TeX is not interested in what it encounters beyond these tokens and this skipping (therefore) goes real fast but it does see nested conditions and doesn't interpret grouping related tokens.

A side effect of this is that the next is not working as expected:

```
\def\ifmorethan{\ifdim\scratchdimen>}
\def\iflessthan{\ifdim\scratchdimen<}

\ifmorethan10pt
  \iflessthan20pt
    result a
  \else
    result b
  \fi
\else
  result c
\fi
```

The `\iflessthan` macro is not seen as an `\if...` so the nesting gets messed up. The solution is to fool the scanner in thinking that it is. Say we have:

```
\scratchdimen=25pt

\def\ifmorethan{\ifdim\scratchdimen>}
\def\iflessthan{\ifdim\scratchdimen<}
```

and:

```
\ifcondition\ifmorethan10pt
  \ifcondition\iflessthan20pt
    result a
  \else
    result b
  \fi
\else
  result c
\fi
```

When we expand this snippet we get: “result b” and no error concerning a failure in locating the right `\fi`'s. So, when scanning the `\ifcondition` is seen as a valid

`\if...` but when the condition is really expanded it gets ignored and the `\ifmorethan` has better come up with a match or not.

In this perspective it is also worth mentioning that nesting problems can be avoided this way:

```
\def\WhenTrue {something \iftrue ...}
\def\WhenFalse{something \iffalse ...}

\ifnum\scratchcounter>123
  \let\next\WhenTrue
\else
  \let\next\WhenFalse
\fi
\next
```

This trick is mentioned in The \TeX book and can also be found in the plain \TeX format. A variant is this:

```
\ifnum\scratchcounter>123
  \expandafter\WhenTrue
\else
  \expandafter\WhenFalse
\fi
```

but using `\expandafter` can be quite intimidating especially when there are multiple in a row. It can also be confusing. Take this: an `\ifcondition` expects the code that follows to produce a test. So:

```
\def\ifwhatever#1%
  {\ifdim#1>10pt
    \expandafter\iftrue
  \else
    \expandafter\iffalse
  \fi}

\ifcondition\ifwhatever{10pt}
  result a
\else
  result b
\fi
```

This will not work! The reason is in the already mentioned fact that when we end up in the greater than 10pt case, the scanner will happily push the `\iftrue` after the `\fi`, which is okay, but when skipping over the `\else` it sees a nested condition without matching `\fi`, which makes it fail. I will spare you a solution with lots of nasty tricks, so here is the clean solution using `\ifcondition`:

```
\def\truecondition {\iftrue}
\def\falsecondition{\iffalse}

\def\ifwhatever#1%
  {\ifdim#1>10pt
    \expandafter\truecondition
  \else
    \expandafter\falsecondition
  \fi}

\ifcondition\ifwhatever{10pt}
  result a
\else
  result b
\fi
```

It will be no surprise that the two macros at the top are predefined in `ConTEXt`. It might be more of a surprise that at the time of this writing the usage in `ConTEXt` of this `\ifcondition` primitive is rather minimal. But that might change.

As a further teaser I'll show another simple one,

```
\def\HowOdd#1{\unless\ifnum\numexpr ((#1):2)*2\relax=\numexpr#1\relax}

\ifcondition\HowOdd{1}very \else not so \fi odd
\ifcondition\HowOdd{2}very \else not so \fi odd
\ifcondition\HowOdd{3}very \else not so \fi odd
```

This renders:

```
very odd
not so odd
very odd
```

The code demonstrates several tricks. First of all we use `\numexpr` which permits more complex arguments, like:

```
\ifcondition\HowOdd{4+1}very \else not so \fi odd
```

```
\ifcondition\HowOdd{2\scratchcounter+9}very \else not so \fi odd
```

Another trick is that we use an integer division (the :) which is an operator supported by LuaMetaTeX.

2.5 LuaMetaTeX primitives

2.5.1 \ifnum and ifdim

These have been extended with a few more operators. For instance, we can use a negation:

```
\ifnum 10 > 5 Y\else N\fi
\ifnum 10 !> 5 Y\else N\fi
```

Results in: YN. A bitwise comparison is possible too:

```
\ifnum "02 & 2 Y\else N\fi
\ifnum "02 & 4 Y\else N\fi
\ifnum "02 !& 8 Y\else N\fi
```

yields: YNY. You can also use the Unicode variants €, ¤, ≠, ≤, ≥, ≠, and ≠.

2.5.2 \iffloat

This is a test for a float, much like a test for a dimen without unit.

2.5.3 \ifabsfloat

This is a test for a float, much like a test for a dimen without unit.

2.5.4 \ifintervalnum

This is a test for equality of two numbers within an interval, as in:

```
\ifintervalnum 1 2 1 Y\else N\fi
\ifintervalnum 1 3 1 Y\else N\fi
\ifintervalnum 100 102 1 Y\else N\fi
\ifintervalnum 100 102 3 Y\else N\fi
```

which results in: YNNY.

2.5.5 `\ifintervalldim`

This is a test for equality of two dimensions within an interval, as in:

```
\ifintervalldim 1pt 2pt 1pt Y\else N\fi
\ifintervalldim 1pt 3pt 1pt Y\else N\fi
\ifintervalldim 100pt 102pt 1pt Y\else N\fi
\ifintervalldim 100pt 102pt 3pt Y\else N\fi
```

We get: YNNY.

2.5.6 `\ifintervalfloat`

This is a test for a float, much like a test for a dimen without unit.

2.5.7 `\ifdimexpression`

This is a boolean checker so the comparison is done as part of the expression, as in:

```
\ifdimexpression{10pt > (4pt + 8pt)}Y\else N\fi
```

2.5.8 `\ifnumexpression`

This is a boolean checker so the comparison is done as part of the expression, as in:

```
\ifnumexpression{10 > (4 + 8)}Y\else N\fi
```

2.5.9 `\ifcmpnum`

This one is part of a set of three tests that all are a variant of a `\ifcase` test. A simple example of the first test is this:

```
\ifcmpnum 123 345 less \or equal \else more \fi
```

The test scans for two numbers, which of course can be registers or expressions, and sets the case value to 0, 1 or 2, which means that you then use the normal `\or` and `\else` primitives for follow up on the test.

2.5.10 `\ifchknum`

This test scans a number and when it's okay sets the case value to 1, and otherwise to 2. So you can do the next:

```
\ifchknum 123\or good \else bad \fi
\ifchknum bad\or good \else bad \fi
```

An error message is suppressed and the first `\or` can be seen as a sort of recovery token, although in fact we just use the fast scanner mode that comes with the `\ifcase`: because the result is 1 or 2, we never see invalid tokens.

In order to avoid another scan the a valid result it is made available in `\lastchknumber`.

2.5.11 `\ifchknumber`

This one is a more rigorous variant of `\ifchknum` and doesn't like trailing non numeric crap.

2.5.12 `\ifchknumexpr`

This test goes a bit further and accepts an expression.

```
\ifchknumexpr 123 + 45\or good \else bad \fi
```

As with the other checkers, if there is a valid result it is available in `\lastchknumber`.

2.5.13 `\ifnumval`

A sort of combination of the previous two is `\ifnumval` which checks a number but also if it's less, equal or more than zero:

```
\ifnumval 123\or less \or equal \or more \else error \fi
\ifnumval bad\or less \or equal \or more \else error \fi
```

You can decide to ignore the bad number or do something that makes more sense. Often the to be checked value will be the content of a macro or an argument like `#1`.

2.5.14 `\ifcmpdim`

This test is like `\ifcmpnum` but for dimensions.

2.5.15 `\ifchkdim`

This test is like `\ifchknum` but for dimensions. The last checked value is available as `\lastchknumber`.

2.5.16 `\ifchkdimension`

This one is a more rigorous variant of `\ifchkdim` and doesn't like trailing rubbish.

2.5.17 `\ifchkdimexpr`

This test is like `\ifchknumexpr` but for dimensions. The last checked value is available as `\lastchkdimension`.

2.5.18 `\ifdimval`

This test is like `\ifnumval` but for dimensions. The last checked value is available as `\lastchkdim`

2.5.19 `\iftok`

Although this test is still experimental it can be used. What happens is that two to be compared ‘things’ get scanned for. For each we first gobble spaces and `\relax` tokens. Then we can have several cases:

1. When we see a left brace, a list of tokens is scanned upto the matching right brace.
2. When a reference to a token register is seen, that register is taken as value.
3. When a reference to an internal token register is seen, that register is taken as value.
4. When a macro is seen, its definition becomes the to be compared value.
5. When a number is seen, the value of the corresponding register is taken

An example of the first case is:

```
\iftok {abc} {def}%
...
\else
...
\fi
```

The second case goes like this:

```
\iftok\scratchtoksone\scratchtokstwo
...
\else
...
\fi
```

Case one and four mixed:

```
\iftok{123}\TempX
...
\else
...
\fi
```

The last case is more a catch: it will issue an error when no number is given. Eventually that might become a bit more clever (depending on our needs.)

2.5.20 \ifzeronum, \ifzerodim, \ifzerofloat

The names of these three tells what they do: checking for a zero value.

```
(\ifzerodim 10pt\norelax A\orelse\ifzerodim 0pt\norelax B\else C\fi)
(\ifzeronum 10 \norelax A\orelse\ifzeronum 0 \norelax B\else C\fi)
(\ifzerofloat 10.0\norelax A\orelse\ifzerofloat 0.0\norelax B\else C\fi)
```

Here we use the \norelax to get rid of trailing spaces: (B) (B) (B).

2.5.21 \ifhaschar, \ifhastok, \ifhastoks, \ifhasxtoks

These checkers can be used to identify a (sequence) of token(s) in a given token list. Their working can best be shown with a few examples:

```
\ifhaschar c {abcd}Y\else N\fi
\ifhastok c {abcd}Y\else N\fi
\ifhastoks {c}{abcd}Y\else N\fi
\ifhasxtoks {c}{abcd}Y\else N\fi

\def\abcd{abcd}

\ifhaschar c {\abcd}Y\else N\fi
\ifhastok c {\abcd}Y\else N\fi
\ifhastoks {c}{\abcd}Y\else N\fi
\ifhasxtoks {c}{\abcd}Y\else N\fi

\ifhaschar c {a{bc}d}Y\else N\fi
\ifhastok c {a{bc}d}Y\else N\fi
\ifhastoks {c}{a{bc}d}Y\else N\fi
\ifhasxtoks {c}{a{bc}d}Y\else N\fi
```

```
\def\abcd{a{bc}d}
```

```
\ifhaschar c {\abcd}Y\else N\fi
```

```
\ifhastok c {\abcd}Y\else N\fi
```

```
\ifhastoks {c}{\abcd}Y\else N\fi
```

```
\ifhasxtoks {c}{\abcd}Y\else N\fi
```

YYYY

NNNY

NYYY

NNNY

The `\ifhaschar` test will not descend into a braced sublist. The `x` variants expand the list before comparison.

2.5.22 `\ifcstok`

There is a subtle difference between this one and `\iftok`: spaces and `\relax` tokens are skipped but nothing gets expanded. So, when we arrive at the to be compared ‘things’ we look at what is there, as-is.

2.5.23 `\iffrozen`

This is an experimental test. Commands can be defined with the `\frozen` prefix and this test can be used to check if that has been the case.

2.5.24 `\ifprotected`

Commands can be defined with the `\protected` prefix (or in `ConTeXt`, for historic reasons, with `\unexpanded`) and this test can be used to check if that has been the case.

2.5.25 `\ifarguments`

This conditional can be used to check how many arguments were matched. It only makes sense when used with macros defined with the `\tolerant` prefix and/or when the sentinel `\ignorearguments` after the arguments is used. More details can be found in the `lowlevel macros manual`.

2.5.26 `\ifrelax`

The following tests all return the same: YYY; it is a shortcut for `\ifx ... \relax` that looks nicer in code.

```

\ifrelax\relax           Y\else N\fi
\ifrelax\norelax        Y\else N\fi
\expandafter\ifrelax\csname ReLaX\endcsname Y\else N\fi

```

2.5.27 `\ifempty`

This is again a shortcut, this time for `\ifx ... \empty` assuming that `\empty` is defined as being nothing. Instead of a token you can also pass a list, so here we get YNY.

```

\ifempty{}              Y\else N\fi
\ifempty{!}             Y\else N\fi
\ifempty\empty          Y\else N\fi

```

2.5.28 `\iflastnamedcs`

This test is part of the `\csname` repertoire and uses the last valid result from such a command.

```

\def\Hello{upper}
\def\hello{lower}
\ifcsname Hello\endcsname
  \iflastnamedcs\hello
    world
  \orelse\iflastnamedcs\Hello
    World
  \fi
\fi

```

Here the ‘Hello’ test result in ‘World’. It is an example of a follow up test, most likely used in user interfacing.

2.5.29 `\ifboolean`

Another new one is the following: it tests a number for being zero or not. As with any primitive that scans for a number, it accepts a braced expression too.

```
(\ifboolean 0 T\else F\fi)
(\ifboolean 1 T\else F\fi)
(\ifboolean {(2 * 4) < 5} T\else F\fi)
(\ifboolean \dimexpression{(1em > 20pt) or (1ex > 15pt)} T\else F\fi)
(\ifboolean \dimexpression{(1em > 3pt) and (1ex < 3pt)} T\else F\fi)
```

We get: (F) (T) (F) (F) (F).

2.5.30 \iflist

The \ifvoid test doesn't really test for a box being empty, which is why we have an additional primitive. Compare the following:

```
\setbox0\hbox{}
\setbox2\hbox{!}
\setbox4\emptybox % \box\voidbox
\setbox8\box6

\wd0 10pt \wd2 10pt \wd4 10pt \wd6 10pt

[\ifvoid0 Y\else N\fi \iflist0 Y\else N\fi \the\wd0] % empty hbox
[\ifvoid2 Y\else N\fi \iflist2 Y\else N\fi \the\wd2] % hbox with content
[\ifvoid4 Y\else N\fi \iflist4 Y\else N\fi \the\wd4] % no box
[\ifvoid6 Y\else N\fi \iflist6 Y\else N\fi \the\wd6] % no box
```

The result demonstrates that we check if there is any content at all, independent of dimensions or the presence of a wrapping list node.

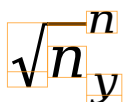
[NN10.0pt] [NY10.0pt] [YN0.0pt] [YN0.0pt]

2.5.31 \ifcramped

This test relates to math and in particular to four of the eight states:

```
\im {
  \sqrt
    {\ifcramped\mathstyle y\else n\fi}
  ^ {\ifcramped\mathstyle y\else n\fi}
  _ {\ifcramped\mathstyle y\else n\fi}
}
```

Because a math formula is first read and then processed in several passes you need to be aware of this state not always being easily predictable because there can be a delay between that read and successive treatments.



2.5.32 `\ifmathparameter`

The next example demonstrates what this test provides:

```
[\ifmathparameter\Umathextrasubspace \displaystyle zero\or set\else
  unset\fi]
[\ifmathparameter\Umathaccentbaseheight\displaystyle zero\or set\else
  unset\fi]
[\ifmathparameter\Umathaccentbasedepth \displaystyle zero\or set\else
  unset\fi]
```

There are three possible outcomes; here we get: [zero] [set] [set]. In LuaMetaT_EX we have more math parameters than in LuaT_EX, and some are set in font specific so called ‘goodie’ files.

2.5.33 `\ifmathstyle`

Here you need to keep in mind that you test the style that is set when T_EX scans for formula. Processing happens afterwards and then styles can change.

```
{\ifmathstyle D\or D'\or T\or T'\or S\or S'\or SS\or SS'\else ?\fi}
\im{\ifmathstyle D\or D'\or T\or T'\or S\or S'\or SS\or SS'\else ?\fi}
\dm{\ifmathstyle D\or D'\or T\or T'\or S\or S'\or SS\or SS'\else ?\fi}
```

We get: ? *T* *D*. The odd values are cramped.

2.5.34 `\iffinalignment`

This test is an experimental one:

```
\halign \bgroup
  \aligncontent
  \alignstab
  \aligncontent
```

```

\cr
one \aligntab \ifinalignment two\else three\fi \cr
\noalign{\ifinalignment yes\else no\fi}
one \aligntab \hbox{\ifinalignment two\else three\fi} \cr
\egroup

\hbox{\ifinalignment two\else three\fi}

```

We get:

```

one two
yes
one two
three

```

2.5.35 \ifinsert

This primitive checks if an insert box has content. Usage depends on the macro package so for instance in ConT_EXt, after `\footnote{A note.}` you can actually check it with:

```
\setupheadertexts[\ifinsert\namedinsertionnumber{footnote} Y\else N\fi]
```

You pass the number of a insert class and in this example the content, set by the page builder, hasn't yet been flushed.

2.5.36 \ifflags

This one related to interfacing. When a macro is defined, one can apply several prefixes to that macro. Some of these prefixes result in a specific kind of macro, for instance a protected, tolerant, tolerant protected, or regular macro. When a macro is defined global, its (internal) level value indicates that. In addition macros, or actually any control sequence, also the built-in ones, can have a set of flags. Some, have consequences in the engine, so for instance an untraced macro will present itself as a primitive, without details that clutter a log. Other flags get meaning when the overload protection mechanisms are enabled.

Testing flags can give some insight but in ConT_EXt there is little reason to use this test other than for illustrative purposes. Take this definition

```
\global\protected\def\Foo{Foo}
```

This macro is internally represented as follows; here we used `\meaning` is:

```
\global \protected \def \Foo {Foo}
```

When we use `\meaning` we get:

```
protected macro:Foo
```

With `\meaningfull` we get:

```
global protected macro:Foo
```

Here is how you can test what properties and flags are set.

```
\ifflags\Foo\global    global    \fi
\ifflags\Foo\protected protected \fi
\ifflags\Foo\tolerant   tolerant   \fi
```

We only show a few tests here:

```
global protected
```

Instead of a prefix you can also pass a number:

```
\ifflags\relax\primitiveflagcode primitive \fi
\ifflags\relax\permanentflagcode permanent \fi
```

```
primitive
```

In `ConTEXt` many macros are defined as permanent which in terms of overload protection has the same impact. Relevant flag values are available in `tex.getflagvalues()` but in `ConTEXt` we prefer predefined constants:

```
\aliasedflagcode, \conditionalflagcode, \constantflagcode, \deferredflag-
code, \frozenflagcode, \globalflagcode, \immediateflagcode, \immutableflag-
code, \inheritedflagcode, \instanceflagcode, \mutableflagcode, \noaligned-
flagcode, \overloadedflagcode, \permanentflagcode, \primitiveflagcode, \pro-
tectedflagcode, \semiprotectedflagcode, \tolerantflagcode, \untracedflag-
code, \valueflagcode
```

2.5.37 `\ifparameters`

This is an `\ifcase` where the number is the number of parameters passed to the current macro. Of course, when used in a macro one should be aware of the fact that another macro call will change this number.

2.5.38 `\ifparameter`

This test checks if a parameter has been set, and it's used as follows:

```
\ifparameter#4\or set\else unset\fi
```

because `#4` is actually a reference it refers to the parameter in the current macro and is not influenced by nested macro calls which makes it more reliable than a `\ifparameters` test.

2.5.39 `\orelse`

This is not really a test primitive but it does act that way. Say that we have this:

```
\ifdim\scratchdimen>10pt
  case 1
\else\ifdim\scratchdimen<20pt
  case 2
\else\ifcount\scratchcounter>10
  case 3
\else\ifcount\scratchcounter<20
  case 4
\fi\fi\fi\fi
```

A bit nicer looks this:

```
\ifdim\scratchdimen>10pt
  case 1
\orelse\ifdim\scratchdimen<20pt
  case 2
\orelse\ifcount\scratchcounter>10
  case 3
\orelse\ifcount\scratchcounter<20
  case 4
\fi
```

We stay at the same level. Sometimes a more flat test tree had advantages but if you think that it gives better performance then you will be disappointed. The fact that we stay at the same level is compensated by a bit more parsing, so unless you have millions such cases (or expansions) it might make a bit of a difference. As mentioned, I'm a bit sensitive for how code looks so that was the main motivation for introducing it. There is a companion `\orunless` continuation primitive.

A rather neat trick is the definition of `\quitcondition`:

```
\def\quitcondition{\orelse\iffalse}
```

This permits:

```
\ifdim\scratchdimen>10pt
  case 1a
  \quitcondition
  case 4b
\fi
```

where, of course, the quitting normally is the result of some intermediate extra test. But let me play safe here: beware of side effects.

2.5.40 `\orunless`

This is the negated variant of `\orelse`.

2.6 For the brave

2.6.1 Full expansion

If you don't understand the following code, don't worry. There is seldom much reason to go this complex but obscure T_EX code attracts some users so ...

When you have a macro that has for instance assignments, and when you expand that macro inside an `\edef`, these assignments are not actually expanded but tokenized. In LuaMetaT_EX there is a way to apply these assignments without side effects and that feature can be used to write a fully expandable user test. For instance:

```
\def\truecondition {\iftrue}
\def\falsecondition{\iffalse}

\def\fontwithidhaschar#1#2%
{\beginlocalcontrol
 \scratchcounter\numexpr\fontid\font\relax
 \setfontid\numexpr#1\relax
 \endlocalcontrol
 \iffontchar\font\numexpr#2\relax
 \beginlocalcontrol
```

```

\setfontid\scratchcounter
\endlocalcontrol
\expandafter\truecondition
\else
\expandafter\falsecondition
\fi}

```

The `\iffontchar` test doesn't handle numeric font id, simply because at the time it was added to ε -T_EX, there was no access to these id's. Now we can do:

```

\edef\foo{\fontwithidhaschar{1} {75}yes\else nop\fi} \meaning\foo
\edef\foo{\fontwithidhaschar{1}{999}yes\else nop\fi} \meaning\foo

[\ifcondition\fontwithidhaschar{1} {75}yes\else nop\fi]
[\ifcondition\fontwithidhaschar{1}{999}yes\else nop\fi]

```

These result in:

```

macro:yes
macro:nop

```

```

[yes]
[nop]

```

If you remove the `\immediateassignment` in the definition above then the typeset results are still the same but the meanings of `\foo` look different: they contain the assignments and the test for the character is actually done when constructing the content of the `\edef`, but for the current font. So, basically that test is now useless.

2.6.2 User defined if's

There is a `\newif` macro that defines three other macros:

```
\newif\ifOnMyOwnTerms
```

After this, not only `\ifOnMyOwnTerms` is defined, but also:

```

\OnMyOwnTermstrue
\OnMyOwnTermsfalse

```

These two actually are macros that redefine `\ifOnMyOwnTerms` to be either equivalent to `\iftrue` and `\iffalse`. The (often derived from plain T_EX) definition of `\newif` is a

bit if a challenge as it has to deal with removing the `if` in order to create the two extra macros and also make sure that it doesn't get mixed up in a catcode jungle.

In ConT_EXt we have a variant:

```
\newconditional\MyConditional
```

that can be used with:

```
\settrue\MyConditional
\setfalse\MyConditional
```

and tested like:

```
\ifconditional\MyConditional
...
\else
...
\fi
```

This one is cheaper on the hash and doesn't need the two extra macros per test. The price is the use of `\ifconditional`, which is *not* to confused with `\ifcondition` (it has bitten me already a few times).

2.7 Relaxing

When T_EX scans for a number or dimension it has to check tokens one by one. On the case of a number, the scanning stops when there is no digit, in the case of a dimension the unit determine the end of scanning. In the case of a number, when a token is not a digit that token gets pushed back. When digits are scanned a trailing space or `\relax` is pushed back. Instead of a number of dimension made from digits, periods and units, the scanner also accepts registers, both the direct accessors like `\count` and `\dimen` and those represented by one token. Take these definitions:

```
\newdimen\MyDimenA \MyDimenA=1pt \dimen0=\MyDimenA
\newdimen\MyDimenB \MyDimenB=2pt \dimen2=\MyDimenB
```

I will use these to illustrate the side effects of scanning. Watch the spaces in the result.

First I show what effect we want to avoid. When second argument contains a number (digits) the zero will become part of it so we actually check `\dimen00` here.

```
\def\whatever#1#2%
```

```
{\ifdim#1=#20\else1\fi}
```

```
\whatever{1pt}{2pt}      [macro:1]
\whatever{1pt}{1pt}      [macro:0]
\whatever{\dimen 0}{\dimen 2} [macro:1]
\whatever{\dimen 0}{\dimen 0} [macro:]
\whatever\MyDimenA\MyDimenB [macro:1]
\whatever\MyDimenA\MyDimenB [macro:1]
```

The solution is to add a space but watch how that one can end up in the result:

```
\def\whatever#1#2%
  {\ifdim#1=#2 0\else1\fi}
```

```
\whatever{1pt}{2pt}      [macro:1]
\whatever{1pt}{1pt}      [macro:0]
\whatever{\dimen 0}{\dimen 2} [macro:1]
\whatever{\dimen 0}{\dimen 0} [macro:0]
\whatever\MyDimenA\MyDimenB [macro:1]
\whatever\MyDimenA\MyDimenB [macro:1]
```

A variant is using `\relax` and this time we get this token retained in the output.

```
\def\whatever#1#2%
  {\ifdim#1=#2\relax0\else1\fi}
```

```
\whatever{1pt}{2pt}      [macro:1]
\whatever{1pt}{1pt}      [macro:\relax 0]
\whatever{\dimen 0}{\dimen 2} [macro:1]
\whatever{\dimen 0}{\dimen 0} [macro:\relax 0]
\whatever\MyDimenA\MyDimenB [macro:1]
\whatever\MyDimenA\MyDimenB [macro:1]
```

A solution that doesn't have side effects of forcing the end of a number (using a space or `\relax` is one where we use expressions. The added overhead of scanning expressions is taken for granted because the effect is what we like:

```
\def\whatever#1#2%
  {\ifdim\dimexpr#1\relax=\dimexpr#2\relax0\else1\fi}
```

```
\whatever{1pt}{2pt}      [macro:1]
\whatever{1pt}{1pt}      [macro:0]
\whatever{\dimen 0}{\dimen 2} [macro:1]
```

```

\whatever{\dimen 0}{\dimen 0} [macro:0]
\whatever\MyDimenA\MyDimenB [macro:1]
\whatever\MyDimenA\MyDimenB [macro:1]

```

Just for completeness we show a more obscure trick: this one hides assignments to temporary variables. Although performance is okay, it is the least efficient one so far.

```

\def\whatever#1#2%
  {\beginlocalcontrol
   \MyDimenA#1\relax
   \MyDimenB#2\relax
   \endlocalcontrol
   \ifdim\MyDimenA=\MyDimenB0\else1\fi}

\whatever{1pt}{2pt} [macro:1]
\whatever{1pt}{1pt} [macro:0]
\whatever{\dimen 0}{\dimen 2} [macro:1]
\whatever{\dimen 0}{\dimen 0} [macro:0]
\whatever\MyDimenA\MyDimenB [macro:1]
\whatever\MyDimenA\MyDimenB [macro:1]

```

It is kind of a game to come up with alternatives but for sure those involve dirty tricks and more tokens (and runtime). The next can be considered a dirty trick too: we use a special variant of `\relax`. When a number is scanned it acts as `\relax`, but otherwise it just is ignored and disappears.

```

\def\whatever#1#2%
  {\ifdim#1=#2\norelax0\else1\fi}

\whatever{1pt}{2pt} [macro:1]
\whatever{1pt}{1pt} [macro:0]
\whatever{\dimen 0}{\dimen 2} [macro:1]
\whatever{\dimen 0}{\dimen 0} [macro:0]
\whatever\MyDimenA\MyDimenB [macro:1]
\whatever\MyDimenA\MyDimenB [macro:1]

```

2.7 Colofon

Author	Hans Hagen
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com contextgarden.net ntg-context@ntg.nl

3 Boxes



low level

TEX

boxes

Contents

3.1	Introduction	47
3.2	Boxes	47
3.3	T _E X primitives	48
3.4	ε-T _E X primitives	51
3.5	LuaT _E X primitives	51
3.6	LuaMetaT _E X primitives	52
3.7	Splitting	60

3.1 Introduction

An average ConT_EXt user will not use the low level box primitives but a basic understanding of how T_EX works doesn't hurt. In fact, occasionally using a box command might bring a solution not easily achieved otherwise, simply because a more high level interface can also be in the way.

The best reference is of course The T_EXbook so if you're really interested in the details you should get a copy of that book. Below I will not go into details about all kind of glues, kerns and penalties, just boxes it is.

This explanation will be extended when I feel the need (or users have questions that can be answered here).

3.2 Boxes

This paragraph of text is made from lines that contain words that themselves contain symbolic representations of characters. Each line is wrapped in a so called horizontal box and eventually those lines themselves get wrapped in what we call a vertical box.

When we expose some details of a paragraph it looks like this:

H	This is a rather narrow	LH:0.000 LB:0.000	RS:13.685	RH:0.000
H	paragraph blown up a	LH:0.000 LB:0.000	RS:19.008	RH:0.000
H	bit. Here we use a flush	LH:0.000 LB:0.000	RS:7.007	RH:0.000
H	left, aka ragged right,	LH:0.000 LB:0.000	RS:19.706	RH:0.000
H	approach.	LH:0.000 LB:0.000	RS:0.000	RH:0.000

The left only shows the boxes, the variant at the right shows (font) kerns and glue too. Because we flush left, there is rather strong right skip glue at the right boundary of the box. If font kerns show up depends on the font, not all fonts have them (or have only a few). The glyphs themselves are also kind of boxed, as their dimensions determine the area that they occupy:

This is a rather ...

But, internally they are not really boxed, as they already are a single quantity. The same is true for rules: they are just blobs with dimensions. A box on the other hand wraps a linked list of so called nodes: glyphs, kerns, glue, penalties, rules, boxes, etc. It is a container with properties like width, height, depth and shift.

3.3 T_EX primitives

The box model is reflected in T_EX's user interface but not by that many commands, most noticeably `\hbox`, `\vbox` and `\vtop`. Here is an example of the first one:

```
\hbox width 10cm{text}
\hbox width 10cm height 1cm depth 5mm{text}
text \raise5mm\hbox{text} text
```

The `\raise` and `\lower` commands behave the same but in opposite directions. One could as well have been defined in terms of the other.

```
text \raise 5mm \hbox to 2cm {text}
text \lower -5mm \hbox to 2cm {text}
text \raise -5mm \hbox to 2cm {text}
text \lower 5mm \hbox to 2cm {text}
```

text. text. text. text.

text text

A box can be moved to the left or right but, believe it or not, in ConT_EXt we never use that feature, probably because the consequences for the width are such that we can as well use kerns. Here are some examples:

```
text \vbox{\moveleft 5mm \hbox {left}}text !
text \vbox{\moveright 5mm \hbox{right}}text !
```

lefttext ! text righttext !

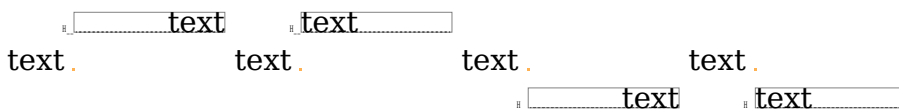
```
text \vbox{\moveleft 25mm \hbox {left}}text !
text \vbox{\moveright 25mm \hbox {right}}text !
```

left text text ! text righttext !

Code like this will produce a complaint about an underfull box but we can easily get around that:

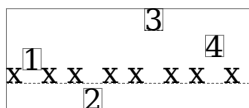
```
text \raise 5mm \hbox to 2cm {\hss text}
text \lower -5mm \hbox to 2cm {text\hss}
text \raise -5mm \hbox to 2cm {\hss text}
text \lower 5mm \hbox to 2cm {text\hss}
```

The `\hss` primitive injects a glue that when needed will fill up the available space. So, here we force the text to the right or left.



Instead of `\raise` you can also provide the shift (up or down) with a keyword:

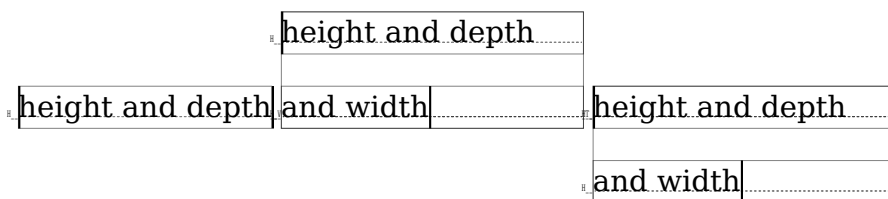
```
\ruledhbox\bggroup
x\raise 5pt\ruledhbox {1}x
x\raise -10pt\ruledhbox {2}x
x\raise -5pt\ruledhbox shift -20pt{3}x
x\ruledhbox shift -10pt{4}x
\egroup
```



We have three kind of boxes: `\hbox`, `\vbox` and `\vtop`. Actually we have a fourth type `\dbox` but that is a variant on `\vbox` to which we come back later.

```
\hbox{\strut height and depth\strut}
\vbox{\hsize 4cm \strut height and depth\par and width\strut}
\vtop{\hsize 4cm \strut height and depth\par and width\strut}
```

A `\vbox` aligns at the bottom and a `\vtop` at the top. I have added some so called struts to enforce a consistent height and depth. A strut is an invisible quantity (consider it a black box) that enforces consistent line dimensions: height and depth.



You can store a box in a register but you need to be careful not to use a predefined one. If you need a lot of boxes you can reserve some for your own:

```
\newbox\MySpecialBox
```

but normally you can do with one of the scratch registers, like 0, 2, 4, 6 or 8, for local boxes, and 1, 3, 5, 7 and 9 for global ones. Registers are used like:

```
\setbox0\hbox{here}
\global\setbox1\hbox{there}
```

In ConT_EXt you can also use

```
\setbox\scratchbox \hbox{here}
\setbox\scratchboxone\hbox{here}
\setbox\scratchboxtwo\hbox{here}
```

and some more. In fact, there are quite some predefined scratch registers (boxes, dimensions, counters, etc). Feel free to investigate further.

When a box is stored, you can consult its dimensions with `\wd`, `\ht` and `\dp`. You can of course store them for later use.

```
\scratchwidth \wd\scratchbox
\scratchheight\ht\scratchbox
\scratchdepth \dp\scratchbox
\scratchtotal \dimexpr\ht\scratchbox+\dp\scratchbox\relax
\scratchtotal \htdp\scratchbox
```

The last line is ConT_EXt specific. You can also set the dimensions

```
\wd\scratchbox 10cm
\ht\scratchbox 10mm
\dp\scratchbox 5mm
```

So you can cheat! A box is placed with `\copy`, which keeps the original intact or `\box` which just inserts the box and then wipes the register. In practice you seldom need a

copy, which is more expensive in runtime anyway. Here we use copy because it serves the examples.

```
\copy\scratchbox
\box \scratchbox
```

3.4 ε -T_EX primitives

The ε -T_EX extensions don't add something relevant for boxes, apart from that you can use the expressions mechanism to mess around with their dimensions. There is a mechanism for typesetting r2l within a paragraph but that has limited capabilities and doesn't change much as it's mostly a way to trick the backend into outputting a stretch of text in the other direction. This feature is not available in LuaT_EX because it has an alternative direction mechanism.

3.5 LuaT_EX primitives

The concept of boxes is the same in LuaT_EX as in its predecessors but there are some aspects to keep in mind. When a box is typeset this happens in LuaT_EX:

1. A list of nodes is constructed. In LuaT_EX this is a double linked list (so that it can easily be manipulated in Lua) but T_EX itself only uses the forward links.
2. That list is hyphenated, that is: so called discretionary nodes are injected. This depends on the language properties of the glyph (character) nodes.
3. Then ligatures are constructed, if the font has such combinations. When this built-in mechanism is used, in ConT_EXt we speak of base mode.
4. After that inter-character kerns are applied, if the font provides them. Again this is a base mode action.
5. Finally the box gets packaged:
 - In the case of a horizontal box, the list is packaged in a hlist node, basically one liner, and its dimensions are calculated and set.
 - In the case of a vertical box, the paragraph is broken into one or more lines, without hyphenation, with optimal hyphenation or in the worst case with so called emergency stretch applied, and the result becomes a vlist node with its dimensions set.

In traditional T_EX the first four steps are interwoven but in LuaT_EX we need them split because the step 5 can be overloaded by a callback. In that case steps 3 and 4 (and maybe 2) are probably also overloaded, especially when you bring handling of fonts under Lua control.

New in LuaT_EX are three packers: `\hpack`, `\vpack` and `\tpack`, which are companions to `\hbox`, `\vbox` and `\vtop` but without the callbacks applied. Using them is a bit tricky as you never know if a callback should be applied, which, because users can often add their own Lua code, is not something predictable.

Another box related extension is direction. There are four possible directions but because in LuaMetaT_EX there are only two. Because this model has been upgraded, it will be discussed in the next section. A ConT_EXt user is supposed to use the official ConT_EXt interfaces in order to be downward compatible.

3.6 LuaMetaT_EX primitives

There are two possible directions: left to right (the default) and right to left for Hebrew and Arabic. Here is an example that shows how it'd done with low level directives:

```
\hbox direction 0 {from left to right}
\hbox direction 1 {from right to left}
```

from left to right
tfel ot thgir morf

A low level direction switch is done with:

```
\hbox direction 0
  {from left to right \textdirection 1 from right to left}
\hbox direction 1
  {from right to left \textdirection 1 from left to right}
```

from left to right tfel ot thgir morf
thgir ot tfel morf tfel ot thgir morf

but actually this is kind of *not done* in ConT_EXt, because there you are supposed to use the proper direction switches:

```
\naturalhbox {from left to right}
\reversehbox {from right to left}
\naturalhbox {from left to right \righttoleft from right to left}
```

`\reversehbox {from right to left \lefttoright from left to right}`

from left to right

tfel ot thgir morf

from left to right tfel ot thgir morf

from left to right tfel ot thgir morf

Often more is needed to properly support right to left typesetting so using the ConT_EXt commands is more robust.

In LuaMetaT_EX the box model has been extended a bit, this as a consequence of dropping the vertical directional typesetting, which never worked well. In previous sections we discussed the properties width, height and depth and the shift resulting from a `\raise`, `\lower`, `\moveleft` and `\moveright`. Actually, the shift is also used in for instance positioning math elements.

The way shifting influences dimensions can be somewhat puzzling. Internally, when T_EX packages content in a box there are two cases:

- When a horizontal box is made, and height - shift is larger than the maximum height so far, that delta is taken. When depth + shift is larger than the current depth, then that depth is adapted. So, a shift up influences the height and a shift down influences the depth.
- In the case of vertical packaging, when width + shift is larger than the maximum box (line) width so far, that maximum gets bumped. So, a shift to the right can contribute, but a shift to the left cannot result in a negative width. This is also why vertical typesetting, where height and depth are swapped with width, goes wrong: we somehow need to map two properties onto one and conceptually T_EX is really set up for horizontal typesetting. (And it's why I decided to just remove it from the engine.)

This is one of these cases where T_EX behaves as expected but it also means that there is some limitation to what can be manipulated. Setting the shift using one of the four commands has a direct consequence when a box gets packaged which happens immediately because the box is an argument to the foursome.

There is in traditional T_EX, probably for good reason, no way to set the shift of a box, if only because the effect would normally be none. But in LuaT_EX we can cheat, and therefore, for educational purposed ConT_EXt has implements some cheats.

We use this sample box:

```

\setbox\scratchbox\hbox\bgroup
  \middlegray\vrule width 20mm depth -.5mm height 10mm
  \hskip-20mm
  \darkgray \vrule width 20mm height -.5mm depth 5mm
\egroup

```

When we mess with the shift using the ConT_EXt `\shiftbox` helper, we see no immediate effect. We only get the shift applied when we use another helper, `\hpackbox`.

```

\hbox\bgroup
  \showstruts \strut
  \quad \copy\scratchbox
  \quad \shiftbox\scratchbox -20mm \copy\scratchbox
  \quad \hpackbox\scratchbox \box \scratchbox
  \quad \strut
\egroup

```



When instead we use `\vpackbox` we get a different result. This time we move left.

```

\hbox\bgroup
  \showstruts \strut
  \quad \copy\scratchbox
  \quad \shiftbox\scratchbox -10mm \copy\scratchbox
  \quad \vpackbox\scratchbox \copy\scratchbox
  \quad \strut
\egroup

```



The shift is set via Lua and the repackaging is also done in Lua, using the low level `hpack` and `vpack` helpers and these just happen to look at the shift when doing their job. At the T_EX end this never happens.

This long exploration of shifting serves a purpose: it demonstrates that there is not that much direct control over boxes apart from their three dimensions. However this was never a real problem as one can just wrap a box in another one and use kerns to move the embedded box around. But nevertheless I decided to see if the engine can be a bit more helpful, if only because all that extra wrapping gives some overhead and complications when we want to manipulate boxes. And of course it is also a nice playground.

We start with changing the direction. Changing this property doesn't require repackaging because directions are not really dealt with in the frontend. When a box is converted to (for instance pdf) the reversion happens.

```
\setbox\scratchbox\hbox{whatever}
\the\boxdirection\scratchbox: \copy\scratchbox \crlf
\boxdirection\scratchbox 1
\the\boxdirection\scratchbox: \copy\scratchbox
```

0: whatever

1: revetahw

Another property that can be queried and set is an attribute. In order to get a private attribute we define one.

```
\newattribute\MyAt
\setbox\scratchbox\hbox attr \MyAt 123 {whatever}
[\the\boxattribute\scratchbox\MyAt]
\boxattribute\scratchbox\MyAt 456
[\the\boxattribute\scratchbox\MyAt]
[\ifnum\boxattribute\scratchbox\MyAt>400 okay\fi]
```

[123] [456] [okay]

The sum of the height and depth is available too. Because for practical reasons setting that property is also needed then, the choice was made to distribute the value equally over height and depth.

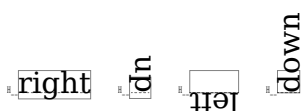
```
\setbox\scratchbox\hbox {height and depth}
[\the\ht\scratchbox]
[\the\dp\scratchbox]
[\the\boxtotal\scratchbox]
\boxtotal\scratchbox=20pt
[\the\ht\scratchbox]
```

```
[\the\dp\scratchbox]
[\the\boxtotal\scratchbox]
```

```
[8.35742pt] [2.44385pt] [10.80127pt] [10.0pt] [10.0pt] [20.0pt]
```

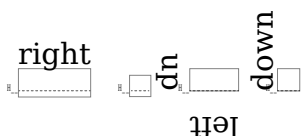
We've now arrived to a set of properties that relate to each other. They are a bit complex and given the number of possibilities one might need to revert to some trial and error: orientations and offsets. As with the dimensions, directions and attributes, they are passed as box specification. We start with the orientation.

```
\hbox \bgroup \showboxes
    \hbox orientation 0 {right}
  \quad \hbox orientation 1 {up}
  \quad \hbox orientation 2 {left}
  \quad \hbox orientation 3 {down}
\egroup
```



When the orientation is set, you can also set an offset. Where shifting around a box can have consequences for the dimensions, an offset is virtual. It gets effective in the backend, when the contents is converted to some output format.

```
\hbox \bgroup \showboxes
    \hbox orientation 0 yoffset 10pt {right}
  \quad \hbox orientation 1 xoffset 10pt {up}
  \quad \hbox orientation 2 yoffset -10pt {left}
  \quad \hbox orientation 3 xoffset -10pt {down}
\egroup
```



The reason that offsets are related to orientation is that we need to know in what direction the offsets have to be applied and this binding forces the user to think about it. You can also set the offsets using commands.

```
\setbox\scratchbox\hbox{whatever}%
1 \copy\scratchbox
2 \boxorientation\scratchbox 2 \copy\scratchbox
```

```

3 \boxxoffset \scratchbox -15pt \copy\scratchbox
4 \boxyoffset \scratchbox -15pt \copy\scratchbox
5

```

```

1 whatever2 3 4 5

```

```

\setbox\scratchboxone\hbox{whatever}%
\setbox\scratchboxtwo\hbox{whatever}%
1 \boxxoffset \scratchboxone -15pt \copy\scratchboxone
2 \boxyoffset \scratchboxone -15pt \copy\scratchboxone
3 \boxxoffset \scratchboxone -15pt \copy\scratchboxone
4 \boxyoffset \scratchboxone -15pt \copy\scratchboxone
5 \boxxmove \scratchboxtwo -15pt \copy\scratchboxtwo
6 \boxymove \scratchboxtwo -15pt \copy\scratchboxtwo
7 \boxxmove \scratchboxtwo -15pt \copy\scratchboxtwo
8 \boxymove \scratchboxtwo -15pt \copy\scratchboxtwo

```

```

whatever 2 3 4 whatever6 7 8
  whatever  whatever  whatever  whatever  whatever

```

The move commands are provided as convenience and contrary to the offsets they do adapt the dimensions. Internally, with the box, we register the orientation and the offsets and when you apply these commands multiple times the current values get overwritten. But ... because an orientation can be more complex you might not get the effects you expect when the options we discuss next are used. The reason is that we store the original dimensions too and these come into play when these other options are used: anchoring. So, normally you will apply an orientation and offsets once only.

The orientation specifier is actually a three byte number that best can be seen hexadecimal (although we stay within the decimal domain). There are three components: x-anchoring, y-anchoring and orientation:

```
0x<X><Y><O>
```

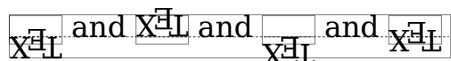
or in TeX speak:

```
"<X><Y><O>
```

The landscape and seascape variants both sit on top of the baseline while the flipped variant has its depth swapped with the height. Although this would be enough a bit more control is possible.

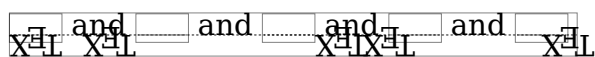
The vertical options of the horizontal variants anchor on the baseline, lower corner, upper corner or center.

```
\ruledhbox orientation "002 {\TEX} and
\ruledhbox orientation "012 {\TEX} and
\ruledhbox orientation "022 {\TEX} and
\ruledhbox orientation "032 {\TEX}
```



The horizontal options of the horizontal variants anchor in the center, left, right, halfway left and halfway right.

```
\ruledhbox orientation "002 {\TEX} and
\ruledhbox orientation "102 {\TEX} and
\ruledhbox orientation "202 {\TEX} and
\ruledhbox orientation "302 {\TEX} and
\ruledhbox orientation "402 {\TEX}
```



The orientation has consequences for the dimensions so they are dealt with in the expected way in constructing lines, paragraphs and pages, but the anchoring is virtual, like the offsets. There are two extra variants for orientation zero: on top of baseline or below, with dimensions taken into account.

```
\ruledhbox orientation "000 {\TEX} and
\ruledhbox orientation "004 {\TEX} and
\ruledhbox orientation "005 {\TEX}
```



The anchoring can look somewhat confusing but you need to keep in mind that it is normally only used in very controlled circumstances and not in running text. Wrapped in macros users don't see the details. We're talking boxes here, so for instance:

```
test\quad
\hbox orientation 3 \bgroup
  \strut test\hbox orientation "002 \bgroup\strut test\egroup test%
\egroup \quad
\hbox orientation 3 \bgroup
  \strut test\hbox orientation "002 \bgroup\strut test\egroup test%
```

```

\egroup \quad
\hbox orientation 3 \bgroup
  \strut test\hbox orientation "012 \bgroup\strut test\egroup test%
\egroup \quad
\hbox orientation 3 \bgroup
  \strut test\hbox orientation "022 \bgroup\strut test\egroup test%
\egroup \quad
\hbox orientation 3 \bgroup
  \strut test\hbox orientation "032 \bgroup\strut test\egroup test%
\egroup \quad
\hbox orientation 3 \bgroup
  \strut test\hbox orientation "042 \bgroup\strut test\egroup test%
\egroup
\quad test

```



Where a `\vtop` has the baseline at the top, a `\vbox` has it at the bottom. In LuaMeta-TeX we also have a `\dbox`, which is a `\vbox` with that behaves like a `\vtop` when it's appended to a vertical list: the height of the first box or rule determines the (base)line correction that gets applied. The following example demonstrates this:

<pre> XXXXXXXXXXXXXXXXXX We thrive in information-thick worlds be- cause of our marvelous and everyday ca- pacity to select, edit, single out, struc- ture, highlight, group, pair, merge, har- monize, synthesize, focus, organize, con- dense, reduce, boil down, choose, cate- gorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discrim- inate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, av- erage, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsisize, winnow the wheat from the chaff and separate the sheep from the goats. XXXXXXXXXXXXXXXXXX </pre>	<pre> XXXXXXXXXXXXXXXXXX We thrive in information-thick worlds be- cause of our marvelous and everyday ca- pacity to select, edit, single out, struc- ture, highlight, group, pair, merge, har- monize, synthesize, focus, organize, con- dense, reduce, boil down, choose, cate- gorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discrim- inate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, av- erage, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsisize, winnow the wheat from the chaff and separate the sheep from the goats. XXXXXXXXXXXXXXXXXX </pre>	<pre> XXXXXXXXXXXXXXXXXX We thrive in information-thick worlds be- cause of our marvelous and everyday ca- pacity to select, edit, single out, struc- ture, highlight, group, pair, merge, har- monize, synthesize, focus, organize, con- dense, reduce, boil down, choose, cate- gorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discrim- inate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, av- erage, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsisize, winnow the wheat from the chaff and separate the sheep from the goats. XXXXXXXXXXXXXXXXXX </pre>
<code>\vbox</code>	<code>\vtop</code>	<code>\dbox</code>

The `d` stands for ‘dual’ because we (sort of) have two baselines. The regular height and depth are those of a `\vbox`.

3.7 Splitting

When you feed $\text{T}_{\text{E}}\text{X}$ a paragraph of text it will accumulate the content in a list of nodes. When the paragraph is finished by `\par` or an empty line it will be fed into the par builder that will try to break the lines as good as possible. Normally that paragraph will be added to the page and at some point there can be breaks between lines in order not to overflow the page. When you collect the paragraph in a box you can use `\vsplit` to emulate this.

```
\setbox\scratchbox\vbox{\samplefile{tufte}}

\startlinecorrection
\rule{0pt}{\vsplit\scratchbox to 2\lineheight}
\stoplinecorrection
```

We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthe-

The split off box is given the specified height, but in $\text{LuaMetaT}_{\text{E}}\text{X}$ you can also get the natural dimensions:

```
\setbox\scratchbox\vbox{\samplefile{tufte}}

\startlinecorrection
\rule{0pt}{\vsplit\scratchbox upto 2\lineheight}
\stoplinecorrection
```

We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthe-

We can force a resulting box type by using `\vsplit`, `\tsplit` and `\dsplit` (here we use the visualized variants):

```
\setbox\scratchbox\vbox{\samplefile{tufte}}

\startlinecorrection
\rule{0pt}{\tsplit\scratchbox upto 2\lineheight}
\stoplinecorrection
```

We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthe-

```
\setbox\scratchbox\vbox{\samplefile{tufte}}
```

```

\startlinecorrection
\ruledvsplit \scratchbox upto 2\lineheight
\stoplinecorrection

```

We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthe-

```
\setbox\scratchbox\vbox{\samplefile{tufte}}
```

```

\startlinecorrection
\ruleddsplit \scratchbox upto 2\lineheight
\stoplinecorrection

```

We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthe-

The engine provides vertical splitters but ConT_EXt itself also has a horizontal one.¹

```

\starttexdefinition Test #1#2#3
  \par
  \dontleavehmode
  \strut
  \llap{{\infofont #2}\quad}
  \blackrule[width=#2,color=darkblue]
  \par
  \setbox\scratchbox\hbox{\samplefile{#1}}
  \hsplit\scratchbox
    to          #2
    depth       \strutdp
    height      \strutht
    shrinkcriterion #3 % badness
  \par
\stoptexdefinition

\dostepwiserecurse {100} {120} {2} {
  \Test{tufte}{#1mm}{1000}
  \Test{tufte}{#1mm}{-100}
}

```

100mm

We thrive in information-thick worlds because of

¹ At some point I might turn that one into a native engine primitive.

100mm We thrive in information-thick worlds because of

102mm We thrive in information-thick worlds because of our

102mm We thrive in information-thick worlds because of our

104mm We thrive in information-thick worlds because of our

104mm We thrive in information-thick worlds because of our

106mm We thrive in information-thick worlds because of our

106mm We thrive in information-thick worlds because of our

108mm We thrive in information-thick worlds because of our

108mm We thrive in information-thick worlds because of our

110mm We thrive in information-thick worlds because of our

110mm We thrive in information-thick worlds because of our

112mm We thrive in information-thick worlds because of our mar-

112mm We thrive in information-thick worlds because of our mar-

114mm We thrive in information-thick worlds because of our mar-

114mm We thrive in information-thick worlds because of our mar-

116mm We thrive in information-thick worlds because of our mar-

116mm We thrive in information-thick worlds because of our mar-

118mm We thrive in information-thick worlds because of our mar-

118mm We thrive in information-thick worlds because of our mar-

120mm We thrive in information-thick worlds because of our mar-

120mm

We thrive in information-thick worlds because of our mar-

A split off box gets packed at its natural size and a badness as well as overshoot amount is calculated. When the overshoot is positive and the the badness is larger than the stretch criterium, the box gets repacked to the natural size. The same happens when the overshoot is negative and the badness exceeds the shrink criterium. When the overshoot is zero (basically we have a fit) but the badness still exceeds the stretch or shrink we also repack. Indeed this is a bit fuzzy, but so is badness.

```
\starttexdefinition Test #1#2#3
  \par
  \dontleavehmode
  \strut
  \llap{{\infofont #2}\quad}
  \blackrule[width=#2,color=darkblue]
  \par
  \setbox\scratchbox\hbox{\samplefile{#1}}
  \doloop {
    \ifvoid\scratchbox
      \exitloop
    \else
      \hsplit\scratchbox
        to      #2
        depth  \strutdp
        height \strutht
        #3
      \par
      \allowbreak
    \fi
  }
\stoptexdefinition

\Test{tufte}{100mm}{shrinkcriterium 1000}
\Test{tufte}{100mm}{shrinkcriterium 0}
\Test{tufte}{100mm}{}

```

100mm

We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge,

harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsise, winnow the wheat from the chaff and separate the sheep from the goats.

100mm

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

100mm

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

BBB
size, winnow the wheat from the chaff and separate
the sheep from the goats.

Watch how the last line get stretched when we set the criterium to zero. I'm sure that users will find reasons to abuse this effect.

3.7 Colofon

Author	Hans Hagen
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com
	contextgarden.net
	ntg-context@ntg.nl

111111

XXX

222222

AAA

4 Expansion



low level

TEX

expansion

Contents

4.1	Preamble	67
4.2	T _E X primitives	67
4.3	ϵ -T _E X primitives	72
4.4	LuaT _E X primitives	74
4.5	LuaMetaT _E X primitives	75
4.6	Dirty tricks	85

4.1 Preamble

This short manual demonstrates a couple of properties of the macro language. It is not an in-depth philosophical expose about macro languages, tokens, expansion and such that some T_EXies like. I prefer to stick to the practical aspects. Occasionally it will be technical but you can just skip those paragraphs (or later return to them) when you can't follow the explanation. It's often not that relevant. I won't talk in terms of mouth, stomach and gut the way the T_EXbook does and although there is no way to avoid the word 'token' I will do my best to not complicate matters by too much token speak. Examples show best what we mean.

4.2 T_EX primitives

The T_EX language provides quite some commands and those built in are called primitives. User defined commands are called macros. A macro is a shortcut to a list of primitives and/or macro calls. All can be mixed with characters that are to be typeset somehow.

```
\def\MyMacro{b}
```

```
a\MyMacro c
```

When T_EX reads this input the a gets turned into a glyph node with a reference to the current font set and the character a. Then the parser sees a macro call, and it will enter another input level where it expands this macro. In this case it sees just an b and it will give this the same treatment as the a. The macro ends, the input level decrements and the c gets its treatment.

Before we move on to more examples and differences between engines, it is good to stress that `\MyMacro` is not a primitive command: we made our command here. The b actually can be seen as a sort of primitive because in this macro it gets stored as so

called token with a primitive property. That primitive property can later on be used to determine what to do. More explicit examples of primitives are `\hbox`, `\advance` and `\relax`. It will be clear that ConT_EXt extends the repertoire of primitive commands with a lot of macro commands. When we typeset a source using module `m-scite` the primitives come out dark blue.

The amount of primitives differs per engine. It all starts with T_EX as written by Don Knuth. Later ε -T_EX added some more primitives and these became official extensions adopted by other variants of T_EX. The pdfT_EX engine added quite some and as follow up on that LuaT_EX added more but didn't add all of pdfT_EX. A few new primitives came from Omega (Aleph). The LuaMetaT_EX engine drops a set of primitives that comes with LuaT_EX and adds plenty new ones. The nature of this engine (no backend and less frontend) makes that we need to implement some primitives as macros. But the basic set is what good old T_EX comes with.

Internally these so called primitives are grouped in categories that relate to their nature. They can be directly expanded (a way of saying that they get immediately interpreted) or delayed (maybe stored for later usage). They can involve definitions, calculations, setting properties and values or they can result in some typesetting. This is what makes T_EX confusing to new users: it is a macro programming language, an interpreter but at the same time an executor of typesetting instructions.

A group of primitives is internally identified as a command (they have a `cmd` code) and the sub commands are flagged by their `chr` code. This sounds confusing but just thing of the fact that most of what we input are characters and therefore they make up most sub commands. For instance the 'letter `cmd`' is used for characters that are seen as letters that can be used in the name of user commands, can be typeset, are valid for hyphenation etc. The letter related `cmd` can have many `chr` codes (all of Unicode). I'd like to remark that the grouping is to a large extend functional, so sometimes primitives that you expect to be similar in nature are in different groups. This has to do with the fact that T_EX needs to be able to determine efficiently if a primitive is operating (or forbidden) in horizontal, vertical and/or math mode.

There are more than 150 internal `cmd` groups. if we forget about the mentioned character related ones, some, have only a few sub commands (`chr`) and others many more (just consider all the OpenType math spacing related parameters). A handful of these commands deal with what we call macros: user defined combinations of primitives and other macros, consider them little programs. The `\MyMacro` example above is an example. There are differences between engines. In standard T_EX there are `\outer` and `\long` commands, and most engines have these. However, in LuaMetaT_EX the later to be discussed `\protected` macros have their own specific 'call `cmd`'. Users don't need to bother about this.

So, when from now on we talk about primitives, we mean the built in, hard coded commands, and when we talk about macros we mean user commands. Although internally there are less cmd categories than primitives, from the perspective of the user they are all unique. Users won't consult the source anyway but when they do they are warned. Also, when in LuaMetaT_EX you use the low level interfacing to T_EX you have to figure out these subtle aspects because there this grouping does matter.

Before we continue I want to make clear that expansion (as discussed in this document) can refer to a macro being expanded (read: its meaning gets injected into the input, so the engine kind of sidetracks from what it was doing) but also to direct consequences of running into a primitive. However, users only need to consider expansion in the perspective of macros. If a user has `\advance` in the input it immediately gets done. But when it's part of a macro definition it only is executed when the macro expands. A good check in (traditional) T_EX is to compare what happens in `\def` and `\edef` which is why we will use these two in the upcoming examples. You put something in a macro and then check what `\meaning` or `\show` reports.

Now back to user defined macros. A macro can contain references to macros so in practice the input can go several levels up and some applications push back a lot so this is why your T_EX input stack can be configured to be huge.

```
\def\MyMacroA{ and }
\def\MyMacroB{1\MyMacroA 2}

a\MyMacroA b
```

When `\MyMacroB` is defined, its body gets three so called tokens: the character token 1 with property 'other', a token that is a reference to the macro `\MyMacroB`, and a character token 2, also with property 'other'. The meaning of `\MyMacroA` is five tokens: a reference to a space token, then three character tokens with property 'letter', and finally a space token.

```
\def \MyMacroA{ and }
\edef\MyMacroB{1\MyMacroA 2}

a\MyMacroA b
```

In the second definition an `\edef` is used, where the e indicates expansion. This time the meaning gets expanded immediately. So we get effectively the same as in:

```
\def\MyMacroB{1 and 2}
```

Characters are easy: they just expand to themselves or trigger adding a glyph node, but not all primitives expand to their meaning or effect.

```
\def\MyMacroA{\scratchcounter = 1 }
\def\MyMacroB{\advance\scratchcounter by 1}
\def\MyMacroC{\the\scratchcounter}
```

```
\MyMacroA a
\MyMacroB b
\MyMacroB c
\MyMacroB d
\MyMacroC
```

a b c d 4

```
macro:\scratchcounter = 1
macro:\advance \scratchcounter by 1
macro:\the \scratchcounter
```

Let's assume that `\scratchcounter` is zero to start with and use `\edef`'s:

```
\edef\MyMacroA{\scratchcounter = 1 }
\edef\MyMacroB{\advance\scratchcounter by 1}
\edef\MyMacroC{\the\scratchcounter}
```

```
\MyMacroA a
\MyMacroB b
\MyMacroB c
\MyMacroB d
\MyMacroC
```

a b c d 0

```
macro:\scratchcounter = 1
macro:\advance \scratchcounter by 1
macro:0
```

So, this time the third macro has its meaning frozen, but we can prevent this by applying a `\noexpand` when we do this:

```
\edef\MyMacroA{\scratchcounter = 1 }
\edef\MyMacroB{\advance\scratchcounter by 1}
\edef\MyMacroC{\noexpand\the\scratchcounter}
```

```
\MyMacroA a
\MyMacroB b
```

```

\MyMacroB c
\MyMacroB d
\MyMacroC

```

```
a b c d 4
```

```

macro:\scratchcounter = 1
macro:\advance \scratchcounter by 1
macro:\the \scratchcounter

```

Of course this is a rather useless example but it serves its purpose: you'd better be aware what gets expanded immediately in an `\edef`. In most cases you only need to worry about `\the` and embedded macros (and then of course their meanings).

You can also store tokens in a so-called token register. Here we use a predefined scratch register:

```

\def\MyMacroA{ and }
\def\MyMacroB{1\MyMacroA 2}
\scratchtoks {\MyMacroA}

```

The content of `\scratchtoks` is: “`\MyMacroA`”, so no expansion has happened here.

```

\def\MyMacroA{ and }
\def\MyMacroB{1\MyMacroA 2}
\scratchtoks \expandafter {\MyMacroA}

```

Now the content of `\scratchtoks` is: “`and`”, so this time expansion has happened.

```

\def\MyMacroA{ and }
\def\MyMacroB{1\MyMacroA 2}
\scratchtoks \expandafter {\MyMacroB}

```

Indeed the macro gets expanded but only one level: “`1\MyMacroA 2`”. Compare this with:

```

\def\MyMacroA{ and }
\edef\MyMacroB{1\MyMacroA 2}
\scratchtoks \expandafter {\MyMacroB}

```

The trick is to expand in two steps with an intermediate `\edef`: “`1 and 2`”. Later we will see that other engines provide some more expansion tricks. The only way to get some grip on expansion is to just play with it.

The `\expandafter` primitive expands the token (which can be a macro) standing after the next next one and then injects its meaning into the stream. So:

```
\expandafter \MyMacroA \MyMacroB
```

works okay. In a normal document you will never need this kind of hackery: it only happens in a bit more complex macros. Here is an example:

```
\scratchcounter 1
\bgroup
\advance\scratchcounter 1
\egroup
\the\scratchcounter
```

```
\scratchcounter 1
\bgroup
\advance\scratchcounter 1
\expandafter
\egroup
\the\scratchcounter
```

The first one gives 1, while the second gives 2.

4.3 ε -T_EX primitives

In this engine a couple of extensions were added and later on pdfT_EX added some more. We only discuss a few that relate to expansion. There is however a pitfall here. Before ε -T_EX showed up, ConT_EXt already had a few mechanism that also related to expansion and it used some names for macros that clash with those in ε -T_EX. This is why we will use the `\normal` prefix here to indicate the primitive.²

```
\def\MyMacroA{a}
\def\MyMacroB{b}
\normalprotected\def\MyMacroC{c}
\edef\MyMacroABC{\MyMacroA\MyMacroB\MyMacroC}
```

These macros have the following meanings:

```
macro:a
macro:b
```

² In the meantime we no longer have a low level `\protected` macro so one can use the primitive

```
protected macro:c
macro:ab\MyMacroC
```

In ConT_EXt you will use the `\unexpanded` prefix instead, because that one did something similar in older versions of ConT_EXt. As we were early adopters of ε -T_EX, this later became a synonym to the ε -T_EX primitive.

```
\def\MyMacroA{a}
\def\MyMacroB{b}
\normalprotected\def\MyMacroC{c}
\normalexpanded{\scratchtoks{\MyMacroA\MyMacroB\MyMacroC}}
```

Here the wrapper around the token register assignment will expand the three macros, unless they are protected, so its content becomes “ab\MyMacroC”. This saves either a lot of more complex `\expandafter` usage or the need to use an intermediate `\edef`. In ConT_EXt the `\expanded` macro does something simpler but it doesn't expand the first token as this is meant as a wrapper around a command, like:

```
\expanded{\chapter{...}} % a ConTEXt command
```

where we do want to expand the title but not the `\chapter` command (not that this would happen actually because `\chapter` is a protected command.)

The counterpart of `\normalexpanded` is `\normalunexpanded`, as in:

```
\def\MyMacroA{a}
\def\MyMacroB{b}
\normalprotected\def\MyMacroC{c}
\normalexpanded {\scratchtoks
  {\MyMacroA\normalunexpanded {\MyMacroB}\MyMacroC}}
```

The register now holds “a\MyMacroB\MyMacroC”: three tokens, one character token and two macro references.

Tokens can represent characters, primitives, macros or be special entities like starting math mode, beginning a group, assigning a dimension to a register, etc. Although you can never really get back to the original input, you can come pretty close, with:

```
\detokenize{this can $ be anything \bgroup}
```

This (when typeset monospaced) is: this can \$ be anything \bgroup. The detokenizer is like `\string` applied to each token in its argument. Compare this to:

```
\normalexpanded {
```

```
\normaldetokenize{10pt}
}
```

We get four tokens: 10pt.

```
\normalexpanded {
  \string 1\string 0\string p\string t
}
```

So that was the same operation: 10pt, but in both cases there is a subtle thing going on: characters have a catcode which distinguishes them. The parser needs to know what makes up a command name and normally that's only letters. The next snippet shows these catcodes:

```
\normalexpanded {
  \noexpand\the\catcode`\string 1 \noexpand\enspace
  \noexpand\the\catcode`\string 0 \noexpand\enspace
  \noexpand\the\catcode`\string p \noexpand\enspace
  \noexpand\the\catcode`\string t \noexpand
}
```

The result is “12 12 11 11”: two characters are marked as ‘letter’ and two fall in the ‘other’ category.

4.4 LuaTeX primitives

This engine adds a little to the expansion repertoire. First of all it offers a way to extend token lists registers:

```
\def\MyMacroA{a}
\def\MyMacroB{b}
\normalprotected\def\MyMacroC{b}
\scratchtoks{\MyMacroA\MyMacroB}
```

The result is: “\MyMacroA\MyMacroB”.

```
\toksapp\scratchtoks{\MyMacroA\MyMacroB}
```

We're now at: “\MyMacroA\MyMacroB\MyMacroA\MyMacroB\MyMacroA\MyMacroB”.

```
\etoksapp\scratchtoks{\MyMacroA\space\MyMacroB\space\MyMacroC}
```

The register has this content: “\MyMacroA\MyMacroB\MyMacroA\MyMacroB a b \MyMacroC a b \MyMacroC”, so the additional context got expanded in the process, except of course the protected macro \MyMacroC.

There is a bunch of these combinators: \toksapp and \tokspre for local appending and prepending, with global companions: \gtoksapp and \gtokspre, as well as expanding variant: \etoksapp, \etokspre, \xtoksapp and \xtokspre.

These are not beforehand more efficient than using intermediate expanded macros or token lists, simply because in the process \TeX has to create token lists too, but sometimes they're just more convenient to use. In \ConTeXt we actually do benefit from these.

4.5 LuaMeta \TeX primitives

We already saw that macro's can be defined protected which means that

```
\def\TestA{A}
\protected \def\TestB{B}
\edef\TestC{\TestA\TestB}
```

gives this:

```
\TestC : A\TestB
```

One way to get \TestB expanded is to prefix it with \expand:

```
\def\TestA{A}
\protected \def\TestB{B}
\edef\TestC{\TestA\TestB}
\edef\TestD{\TestA\expand\TestB}
```

We now get:

```
\TestC : A\TestB
\TestD : AB
```

There are however cases where one wishes this to happen automatically, but that will also make protected macros expand which will create havoc, like switching fonts.

```
\def\TestA{A}
\protected \def\TestB{B}
\semiprotected \def\TestC{C}
\edef\TestD{\TestA\TestB\TestC}
```

```

\edef\TestE{\normalexpanded{\TestA\TestB\TestC}}
\edef\TestF{\semiexpanded {\TestA\TestB\TestC}}

```

This time `\TestC` loses its protection:

```

\TestA : A
\TestB : B
\TestC : C
\TestD : A\TestB \TestC
\TestE : A\TestB \TestC
\TestF : A\TestB C

```

Actually adding `\fullyexpanded` would be trivial but it makes not much sense to add the overhead (at least not now). This feature is experimental anyway so it might go away when I see no real advantage from it.

When you store something in a macro or token register you always need to keep an eye on category codes. A dollar in the input is normally treated as math shift, a hash indicates a macro parameter or preamble entry. Characters like ‘A’ are letters but ‘[’ and ‘]’ are tagged as ‘other’. The \TeX scanner acts according to these codes. If you ever find yourself in a situation that changing catcodes is no option or cumbersome, you can do this:

```

\edef\TestOA{\expandtoken\othercatcode `A}
\edef\TestLA{\expandtoken\lettercatcode `A}

```

In both cases the meaning is A but in the first case it's not a letter but a character flagged as ‘other’.

A whole new category of commands has to do with so called local control. When \TeX scans and interprets the input, a process takes place that is called tokenizing: (sequences of) characters get a symbolic representation and travel through the system as tokens. Often they immediately get interpreted and are then discarded. But when for instance you define a macro they end up as a linked list of tokens in the macro body. We already saw that expansion plays a role. In most cases, unless \TeX is collecting tokens, the main action is dealt with in the so-called main loop. Something gets picked up from the input but can also be pushed back, for instance because of some lookahead that didn't result in an action. Quite some time is spent in pushing and popping from the so-called input stack.

When we are in Lua, we can pipe back into the engine but all is collected till we're back in \TeX where the collected result is pushed into the input. Because \TeX is a mix of programming and action there basically is only that main loop. There is no real way

to start a sub run in Lua and do all kind of things independent of the current one. This makes sense when you consider the mix: it would get too confusing.

However, in LuaTeX and even better in LuaMetaTeX, we can enter a sort of local state and this is called ‘local control’. When we are in local control a new main loop is entered and the current state is temporarily forgotten: we can for instance expand where one level up expansion was not done. It sounds complicated and indeed it is complicated so examples have to clarify it.

```
1 \setbox0\hbox to 10pt{2} \count0=3 \the\count0 \multiply\count0 by 4
```

This snippet of code is not that useful but illustrates what we're dealing with:

- The 1 gets typeset. So, characters like that are seen as text.
- The \setbox primitive triggers picking up a register number, then goes on scanning for a box specification and that itself will typeset a sequence of whatever until the group ends.
- The count primitive triggers scanning for a register number (or reference) and then scans for a number; the equal sign is optional.
- The the primitive injects some value into the current input stream and it does so by entering a new input level.
- The multiply primitive picks up a register specification and multiplies that by the next scanned number. The by is optional.

We now look at this snippet again but with an expansion context:

```
\def \TestA{1 \setbox0\hbox{2} \count0=3 \the\count0}
\edef\TestB{1 \setbox0\hbox{2} \count0=3 \the\count0}
```

These two macros have a slightly different body. Make sure you see the difference before reading on.

control sequence: TestA

332735	12	49	other char	1	U+00031	
587500	10	32	spacer			
592454	129	0	set box			setbox
54255	12	48	other char	0	U+00030	
591377	31	14	make box			hbox
587487	1	123	left brace			

593499	12	50	other char	2	U+00032	
592589	2	125	right brace			
478928	10	32	spacer			
578534	122	1	register			count
592537	12	48	other char	0	U+00030	
587587	12	61	other char	=	U+0003D	
332681	12	51	other char	3	U+00033	
593447	10	32	spacer			
592740	140	0	the			the
332743	122	1	register			count
592583	12	48	other char	0	U+00030	

control sequence: TestB

593876	12	49	other char	1	U+00031	
593097	10	32	spacer			
593705	129	0	set box			setbox
593878	12	48	other char	0	U+00030	
593509	31	14	make box			hbox
593193	1	123	left brace			
593435	12	50	other char	2	U+00032	
593409	2	125	right brace			
536415	10	32	spacer			
478968	122	1	register			count
479267	12	48	other char	0	U+00030	
478981	12	61	other char	=	U+0003D	
592690	12	51	other char	3	U+00033	
593399	10	32	spacer			
593931	12	49	other char	1	U+00031	

We now introduce a new primitive `\localcontrolled`:

```
\edef\TestB{1 \setbox0\hbox{2} \count0=3 \the\count0}
```

```
\edef\TestC{1 \setbox0\hbox{2} \localcontrolled{\count0=3} \the\count0}
```

Again, watch the subtle differences:

control sequence: TestB

593034	12	49	other char	1	U+00031	
592413	10	32	spacer			
478933	129	0	set box			setbox

593153	12	48	other char	0	U+00030	
107844	31	14	make box			hbox
587585	1	123	left brace			
592790	12	50	other char	2	U+00032	
592556	2	125	right brace			
593869	10	32	spacer			
593073	122	1	register			count
593886	12	48	other char	0	U+00030	
593112	12	61	other char	=	U+0003D	
593914	12	51	other char	3	U+00033	
579119	10	32	spacer			
579076	12	49	other char	1	U+00031	

control sequence: TestC

592348	12	49	other char	1	U+00031	
593876	10	32	spacer			
593097	129	0	set box			setbox
593705	12	48	other char	0	U+00030	
593878	31	14	make box			hbox
593509	1	123	left brace			
593193	12	50	other char	2	U+00032	
593435	2	125	right brace			
593409	10	32	spacer			
593399	10	32	spacer			
591506	12	51	other char	3	U+00033	

Another example:

```
\edef\TestB{1 \setbox0\hbox{2} \count0=3 \the\count0}
```

```
\edef\TestD{\localcontrolled{1 \setbox0\hbox{2} \count0=3 \the\count0}}
```

1 3 ← Watch how the results end up here!

control sequence: TestB

593046	12	49	other char	1	U+00031	
479083	10	32	spacer			
593102	129	0	set box			setbox
593427	12	48	other char	0	U+00030	
593183	31	14	make box			hbox
592750	1	123	left brace			

593894	12	50	other char	2	U+00032
593336	2	125	right brace		
592817	10	32	spacer		
592662	122	1	register		count
593428	12	48	other char	0	U+00030
478983	12	61	other char	=	U+0003D
593645	12	51	other char	3	U+00033
592923	10	32	spacer		
593237	12	51	other char	3	U+00033

control sequence: TestD

<no tokens>

We can use this mechanism to define so called fully expandable macros:

```
\def\WidthOf#1%
  {\beginlocalcontrol
   \setbox0\hbox{#1}%
   \endlocalcontrol
   \wd0 }

\scratchdimen\WidthOf{The Rite Of Spring}
```

```
\the\scratchdimen
```

104.72021pt

When you want to add some grouping, it quickly can become less pretty:

```
\def\WidthOf#1%
  {\dimexpr
   \beginlocalcontrol
     \begingroup
       \setbox0\hbox{#1}%
       \expandafter
     \endgroup
     \expandafter
   \endlocalcontrol
   \the\wd0
  \relax}

\scratchdimen\WidthOf{The Rite Of Spring}
```

```
\the\scratchdimen
```

```
104.72021pt
```

A single token alternative is available too and its usage is like this:

```
\def\TestA{\scratchcounter=100 }
\edef\TestB{\localcontrol\TestA \the\scratchcounter}
\edef\TestC{\localcontrolled{\TestA} \the\scratchcounter}
```

The content of `\TestB` is ‘100’ and of course the `\TestC` macro gives ‘ 100’.

We now move to the Lua end. Right from the start the way to get something into $\mathrm{T}_{\mathrm{E}}\mathrm{X}$ from Lua has been the print functions. But we can also go local (immediate). There are several methods:

- via a set token register
- via a defined macro
- via a string

Among the things to keep in mind are catcodes, scope and expansion (especially in when the result itself ends up in macros). We start with an example where we go via a token register:

```
\toks0={\setbox0\hbox{The Rite Of Spring}}
\toks2={\setbox0\hbox{The Rite Of Spring!}}

\startluacode
tex.runlocal(0) context("[1: %p]",tex.box[0].width)
tex.runlocal(2) context("[2: %p]",tex.box[0].width)
\stopluacode
```

```
[1: 104.72021pt][2: 109.14062pt]
```

We can also use a macro:

```
\def\TestA{\setbox0\hbox{The Rite Of Spring}}
\def\TestB{\setbox0\hbox{The Rite Of Spring!}}

\startluacode
tex.runlocal("TestA") context("[3: %p]",tex.box[0].width)
tex.runlocal("TestB") context("[4: %p]",tex.box[0].width)
\stopluacode
```

[3: 104.72021pt][4: 109.14062pt]

A third variant is more direct and uses a (Lua) string:

\startluacode

```
tex.runstring([[ \setbox0\hbox{The Rite Of Spring} ]])
```

```
context("[5: %p]", tex.box[0].width)
```

```
tex.runstring([[ \setbox0\hbox{The Rite Of Spring!} ]])
```

```
context("[6: %p]", tex.box[0].width)
```

\stopluacode

[5: 104.72021pt][6: 109.14062pt]

A bit more high level:

```
context.runstring([[ \setbox0\hbox{(Here \bf 1.2345)} ]])
```

```
context.runstring([[ \setbox0\hbox{(Here \bf %.3f)} ]], 1.2345)
```

Before we had runstring this was the way to do it when staying in Lua was needed:

\startluacode

```
token.setmacro("TestX", [[ \setbox0\hbox{The Rite Of Spring} ]])
```

```
tex.runlocal("TestX")
```

```
context("[7: %p]", tex.box[0].width)
```

\stopluacode

[7: 104.72021pt]

\startluacode

```
tex.scantoks(0, tex.ctxcatcodes, [[ \setbox0\hbox{The Rite Of Spring!} ]])
```

```
tex.runlocal(0)
```

```
context("[8: %p]", tex.box[0].width)
```

\stopluacode

[8: 109.14062pt]

The order of flushing matters because as soon as something is not stored in a token list or macro body, \TeX will typeset it. And as said, a lot of this relates to pushing stuff into the input which is stacked. Compare:

\startluacode

```
context("[HERE 1]")
context("[HERE 2]")
\stopluacode
```

[HERE 1][HERE 2]

with this:

```
\startluacode
tex.pushlocal() context("[HERE 1]") tex.poplocal()
tex.pushlocal() context("[HERE 2]") tex.poplocal()
\stopluacode
```

[HERE 1][HERE 2]

You can expand a macro at the Lua end with `token.expandmacro` which has a peculiar interface. The first argument has to be a string (the name of a macro) or a userdata (a valid macro token). This macro can be fed with parameters by passing more arguments:

string	serialized to tokens
true	wrap the next string in curly braces
table	each entry will become an argument wrapped in braces
token	inject the token directly
number	change control to the given catcode table

There are more scanner related primitives, like the ε -TeX primitive `\detokenize`:

```
[\detokenize {test \relax}]
```

This gives: `[test \relax]`. In LuaMetaTeX we also have complementary primitive(s):

```
[\tokenized    catcodetable \vrbcatcodes {test {\bf test} test}]
[\tokenized                                {test {\bf test} test}]
[\retokenized                \vrbcatcodes {test {\bf test} test}]
```

The `\tokenized` takes an optional keyword and the examples above give: `[test {\bf test} test [test test test] [test {\bf test} test]`. The LuaTeX primitive `\scantextokens` which is a variant of ε -TeX's `\scantokens` operates under the current catcode regime (the last one honors `\everyeof`). The difference with `\tokenized` is that this one first serializes the given token list (just like `\detokenize`).³

³ The `\scan *tokens` primitives now share the same helpers as Lua, but they should behave the same as in LuaTeX.

With `\retokenized` the catcode table index is mandatory (it saves a bit of scanning and is easier on intermixed `\expandafter` usage. There often are several ways to accomplish the same:

```
\def\MyTitle{test {\bf test} test}
\detokenize           \expandafter{\MyTitle}: 0.46\crlf
\meaningless         \MyTitle : 0.47\crlf
\retokenized         \notcatcodes{\MyTitle}: 0.87\crlf
\tokenized   catcodetable \notcatcodes{\MyTitle}: 0.93\crlf
```

```
test {\bf test} test: 0.46
test {\bf test} test: 0.47
test {\bf test} test: 0.87
test {\bf test} test: 0.93
```

Here the numbers show the relative performance of these methods. The `\detokenize` and `\meaningless` win because they already know that a verbose serialization is needed. The last two first serialize and then reinterpret the resulting token list using the given catcode regime. The last one is slowest because it has to scan the keyword.

There is however a pitfall here:

```
\def\MyText {test}
\def\MyTitle{test \MyText\space test}
\detokenize           \expandafter{\MyTitle}\crlf
\meaningless         \MyTitle \crlf
\retokenized         \notcatcodes{\MyTitle}\crlf
\tokenized   catcodetable \notcatcodes{\MyTitle}\crlf
```

The outcome is different now because we have an expandable embedded macro call. The fact that we expand in the last two primitives is also the reason why they are ‘slower’.

```
test \MyText \space test
test \MyText \space test
test test test
test test test
```

To complete this picture, we show a variant than combines much of what has been introduced in this section:

```

\semiprotected\def\MyTextA {test}
\def\MyTextB {test}
\def\MyTitle{test \MyTextA\space \MyTextB\space test}
\detokenize          \expandafter{\MyTitle}\crlf
\meaningless          \MyTitle \crlf
\retokenized          \notcatcodes{\MyTitle}\crlf
\retokenized          \notcatcodes{\semiexpanded{\MyTitle}}\crlf
\tokenized    catcodetable \notcatcodes{\MyTitle}\crlf
\tokenized    catcodetable \notcatcodes{\semiexpanded{\MyTitle}}

```

This time compare the last four lines:

```

test \MyTextA \space \MyTextB \space test
test \MyTextA \space \MyTextB \space test
test \MyTextA test test
test test test test
test \MyTextA test test
test test test test

```

Of course the question remains to what extend we need this and eventually will apply in ConT_EXt. The `\detokenize` is used already. History shows that eventually there is a use for everything and given the way LuaMetaT_EX is structured it was not that hard to provide the alternatives without sacrificing performance or bloating the source.

4.6 Dirty tricks

When I was updating this manual Hans vd Meer and I had some discussions about expansion and tokenization related issues when combining of xml processing with T_EX macros where he did some manipulations in Lua. In these mixed cases you can run into catcode related problems because in xml you want for instance a `#` to be a hash mark (other character) and not an parameter identifier. Normally this is handled well in ConT_EXt but of course there are complex cases where you need to adapt.

Say that you want to compare two strings (officially we should say token lists) with mixed catcodes. Let's also assume that you want to use the normal `\if` construct (which was part of the discussion). We start with defining a test set. The reason that we present this example here is that we use commands discussed in previous sections:

```

\def\abc{abc}
\semiprotected \def\xyz{xyz}
\edef\pqr{\expandtoken\notcatcodes`p%

```

```
\expandtoken\notcatcodes`q%
\expandtoken\notcatcodes`r}
```

```
1: \ifcondition\similartokens{abc} {def}YES\else NOP\fi (NOP) \quad
2: \ifcondition\similartokens{abc}{\abc}YES\else NOP\fi (YES)

3: \ifcondition\similartokens{xyz} {pqr}YES\else NOP\fi (NOP) \quad
4: \ifcondition\similartokens{xyz}{\xyz}YES\else NOP\fi (YES)

5: \ifcondition\similartokens{pqr} {pqr}YES\else NOP\fi (YES) \quad
6: \ifcondition\similartokens{pqr}{\pqr}YES\else NOP\fi (YES)
```

So, we have a mix of expandable and semi expandable macros, and also a mix of cat-codes. A naive approach would be:

```
\permanent\protected\def\similartokens#1#2%
{\iftok{#1}{#2}}
```

but that will fail on some cases:

```
1: NOP(NOP)    2: YES(YES)
3: NOP(NOP)    4: NOP(YES)
5: YES(YES)    6: NOP(YES)
```

So how about:

```
\permanent\protected\def\similartokens#1#2%
{\iftok{\detokenize{#1}}{\detokenize{#2}}}
```

That one is even worse:

```
1: NOP(NOP)    2: NOP(YES)
3: NOP(NOP)    4: NOP(YES)
5: YES(YES)    6: NOP(YES)
```

We need to expand so we end up with this:

```
\permanent\protected\def\similartokens#1#2%
{\normalexpanded{\noexpand\iftok
  {\noexpand\detokenize{#1}}
  {\noexpand\detokenize{#2}}}}
```

Better:

```
1: NOP(NOP)    2: YES(YES)
```

3: NOP(NOP) 4: NOP(YES)
 5: YES(YES) 6: YES(YES)

But that will still not deal with the mildly protected macro so in the end we have:

```
\permanent\protected\def\similartokens#1#2%
  {\semiexpanded{\noexpand\iftok
    {\noexpand\detokenize{#1}}
    {\noexpand\detokenize{#2}}}}
```

Now we're good:

1: NOP(NOP) 2: YES(YES)
 3: NOP(NOP) 4: YES(YES)
 5: YES(YES) 6: YES(YES)

Finally we wrap this one in the usual `\doifelse...` macro:

```
\permanent\protected\def\doifelsesimilartokens#1#2%
  {\ifcondition\similartokens{#1}{#2}%
    \expandafter\firstoftwoarguments
  \else
    \expandafter\secondoftwoarguments
  \fi}
```

so that we can do:

```
\doifelsesimilartokens{pqr}{\pqr}{YES}{NOP}
```

A companion macro of this is `\wipetoken` but for that one you need to look into the source.

4.6 Colofon

Author	Hans Hagen
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com contextgarden.net ntg-context@ntg.nl

5 Registers

low level

TEX

registers

Contents

5.1	Preamble	89
5.2	T _E X primitives	89
5.3	ϵ -T _E X primitives	92
5.4	LuaT _E X primitives	92
5.5	LuaMetaT _E X primitives	93
5.6	Units	94

5.1 Preamble

Registers are sets of variables that are accessed by index and a such resemble registers in a processing unit. You can store a quantity in a register, retrieve it, and also manipulate it.

There is hardly any need to use them in ConT_EXt so we keep it simple.

5.2 T_EX primitives

There are several categories:

- Integers (int): in order to be portable (at the time it surfaced) there are only integers and no floats. The only place where T_EX uses floats internally is when glue gets effective which happens in the backend.
- Dimensions (dimen): internally these are just integers but when they are entered they are sliced into two parts so that we have a fractional part. The internal representation is called a scaled point.
- Glue (skip): these are dimensions with a few additional properties: stretch and shrink. Being a compound entity they are stored differently and thereby a bit less efficient than numbers and dimensions.
- Math glue (muskip): this is the same as glue but with a unit that adapts to the current math style properties. It's best to think about them as being relative measures.
- Token lists (toks): these contain a list of tokens coming from the input or coming from a place where they already have been converted.

The original T_EX engine had 256 entries per set. The first ten of each set are normally reserved for scratch purposes: the even ones for local use, and the odd ones for global

usage. On top of that macro packages can reserve some for its own use. It was quite easy to reach the maximum but there were tricks around that. This limitation is no longer present in the variants in use today.

Let's set a few dimension registers:

```
\dimen 0 = 10 pt
\dimen2=10pt
\dimen4 10pt
\scratchdimen 10pt
```

We can serialize them with:

```
\the \dimen0
\number \dimen2
\meaning\dimen4
\meaning\scratchdimen
```

The results of these operations are:

```
10.0pt
655360
\dimen4
constant dimension 10.0pt
```

The last two is not really useful but it is what you see when tracing options are set. Here `\scratchdimen` is a shortcut for a register. This is *not* a macro but a defined register. The low level `\dimendef` is used for this but in a macro package you should not use that one but the higher level `\newdimen` macro that uses it.

```
\newdimen\MyDimenA
\def \MyDimenB{\dimen999}
\dimendef\MyDimenC998

\meaning\MyDimenA
\meaning\MyDimenB
\meaning\MyDimenC
```

Watch the difference:

```
\dimen269
macro:\dimen 999
\dimen998
```

The first definition uses a yet free register so you won't get a clash. The second one is just a shortcut using a macro and the third one too but again direct shortcut. Try to imagine how the second line gets interpreted:

```
\MyDimenA10pt \MyDimenA10.5pt
\MyDimenB10pt \MyDimenB10.5pt
\MyDimenC10pt \MyDimenC10.5pt
```

Also try to imagine what messing around with `\MyDimenC` will do when we also have defined a few hundred extra dimensions with `\newdimen`.

In the case of dimensions the `\number` primitive will make the register serialize as scaled points without unit `sp`.

Next we see some of the other registers being assigned:

```
\count 0 = 100
\skip 0 = 10pt plus 3pt minus 2pt
\skip 0 = 10pt plus 1fill
\muskip 0 = 10mu plus 3mu minus 2mu
\muskip 0 = 10mu minus 1 fil
\toks 0 = {hundred}
```

When a number is expected, you can use for instance this:

```
\scratchcounter\scratchcounterone
```

Here we use a few predefined scratch registers. You can also do this:

```
\scratchcounter\numexpr\scratchcounterone+\scratchcountertwo\relax
```

There are some quantities that also qualify as number:

```
\chardef\MyChar=123 % refers to character 123 (if present)
\scratchcounter\MyChar
```

In the past using `\chardef` was a way to get around the limited number of registers, but it still had (in traditional \TeX) a limitation: you could not go beyond 255. The `\mathchardef` could go higher as it also encodes a family number and class. This limitation has been lifted in \LuaTeX .

A character itself can also be interpreted as number, in which case it has to be prefixed with a reverse quote: ```, so:

```
\scratchcounter\numexpr`0+5\relax
\char\scratchcounter
```

produces “5” because the ``0` expands into the (ascii and utf8) slot 48 which represents the character zero. In this case the next makes more sense:

```
\char\numexpr`0+5\relax
```

If you want to know more about all these quantities, “T_EX By Topic” provides a good summary of what T_EX has to offer, and there is no need to repeat it here.

5.3 ε -T_EX primitives

Apart from the ability to use expressions, the contribution to registers that ε -T_EX brought was that suddenly we could use upto 65K of them, which is more than enough. The extra registers were not as efficient as the first 256 because they were stored in the hash table, but that was not really a problem. In Omega and later LuaT_EX regular arrays were used, at the cost of more memory which in the meantime has become cheap. As ConT_EXt moved to ε -T_EX rather early its users never had to worry about it.

5.4 LuaT_EX primitives

The LuaT_EX engine introduced attributes. These are numeric properties that are bound to the nodes that are the result of typesetting operations. They are basically like integer registers but when set their values get bound and when unset they are kind of invisible.

- Attribute (attribute): a numeric property that when set becomes part of the current attribute list that gets assigned to nodes.

Attributes can be used to communicate properties to Lua callbacks. There are several functions available for setting them and querying them.

```
\attribute999 = 123
```

Using attributes this way is dangerous (of course I can only speak for ConT_EXt) because an attribute value might trigger some action in a callback that gives unwanted side effects. For convenience ConT_EXt provides:

```
\newattribute\MyAttribute
```

Which currently defines `\MyAttribute` as constant integer 1026 and is meant to be used as:⁴

```
\attribute\MyAttribute = 123
```

Just be aware that defining attributes can have an impact on performance. As you cannot access them at the \TeX end you seldom need them. If you do you can better use the proper more high level definers (not discussed here).

5.5 LuaMeta \TeX primitives

The fact that scanning stops at a non-number or `\relax` can be sort of unpredictable which is why in LuaMeta \TeX we also support the following variant:

```
\scratchdimen\dimexpr 10pt + 3pt \relax
\scratchdimen\dimexpr {10pt + 3pt}
```

At the cost of one more token braces can be used as boundaries instead of the single `\relax` boundary.

An important property of registers is that they can be accessed by a number. This has big consequences for the implementation: they are part of the big memory store and consume dedicated ranges. If we had only named access \TeX 's memory layout could be a bit leaner. In principle we could make the number of registers smaller because any limit on the amount at some point can be an obstacle. It is for that reason that we also have name-only variants:

```
\dimensiondef \MyDimenA 12pt
\integerdef \MyIntegerA 12
\gluespecdef \MyGlueA 12pt + 3pt minus 4pt
\mulgluespecdef \MyMuA 12mu + 3mu minus 4mu
```

These are as efficient but not accessible by number but they behave like registers which means that you (can) use `\the`, `\advance`, `\multiply` and `\divide` with them.⁵ In case you wonder why there is no alternative for `\toksdef`, there actually are multiple: they are called macros.

todo: expressions

⁴ The low level `\attributedef` command is rather useless in the perspective of Con \TeX t.

⁵ There are also the slightly more efficient `\advanceby`, `\multiplyby` and `\divideby` that don't check for the by keyword.

5.6 Units

The LuaMetaT_EX engine supports the following units. The first batch is constant with hard coded fine tuned values. The second set is related to the current font. The last group is kind of special, the es is a replacement for the in and has a little sister in ts. The dk is dedicated to the master and makes a nice offset for so called T_EX pages that we use for demos.

pt	1.0	point
bp	1.00374	big point (aka postscript point)
in	72.26999	inch
cm	28.45274	centimeter
mm	2.84526	milimeter
dd	1.07	didot
cc	12.8401	cicero
pc	12.0	pica
sp	0.00002	scaled points
px	0.00002	pixel

ex	5.70947	ex height
em	11.0	em width
mu	1.0	math unit

ts	7.11317	tove
es	71.13177	edith
eu	71.13177	european unit
dk	6.43985	knuth

The fi[lll] unit is not really a unit but a multiplier for infinite stretch and shrink; original T_EX doesn't have the simple fi.

In addition to these we can have many more. In principle a user can define additional ones but there's always a danger of clashing. For users we reserve the units starting with an u. Here is how you define your own, we show three variants:

```
\newdimension \FooA    \FooA 1.23pt
\newdimen     \FooB    \FooB 12.3pt
\protected\def\FooC    {\the\dimexpr\FooA +\FooB\relax}

\pushoverloadmode % just in case
  \newuserunit\FooA ua
  \newuserunit\FooB ub
  \newuserunit\FooC uc
```

\popoverloadmode

And this is how they show up:

2.45999pt 24.6pt 27.06pt

with

```
\the\dimexpr 2 ua \relax\quad
\the\dimexpr 2 ub \relax\quad
\the\dimexpr 2 uc \relax
```

The following additional units are predefined (reserved). The values are in points and some depend on the current layout and document font.

pi	3.14159	π for Mikael
ft	867.23999	foot for Alan
fs	11.0	(global body) font size
tw	483.69687	(layout) text width
th	645.88272	(layout) text height
hs	483.69687	(current) hsize
vs	645.88272	(current) vsize
cd	0.0	(when set) column distance
cw	483.69687	(when set) column width
cx	236.34843	combination cell width
uu	28.45274	user unit (MetaFun)
fw	0.0	framed width
fh	0.0	framed height
fo	0.0	framed offset
lw	0.4	line width
sh	11.51031	strut height
sd	4.47621	strut depth
st	15.98653	strut total
ch	6.99854	width of zero (css)
fa	8.35742	font ascender
fd	1.71338	font descender
fc	8.01904	font cap height

Here is an example of usage:

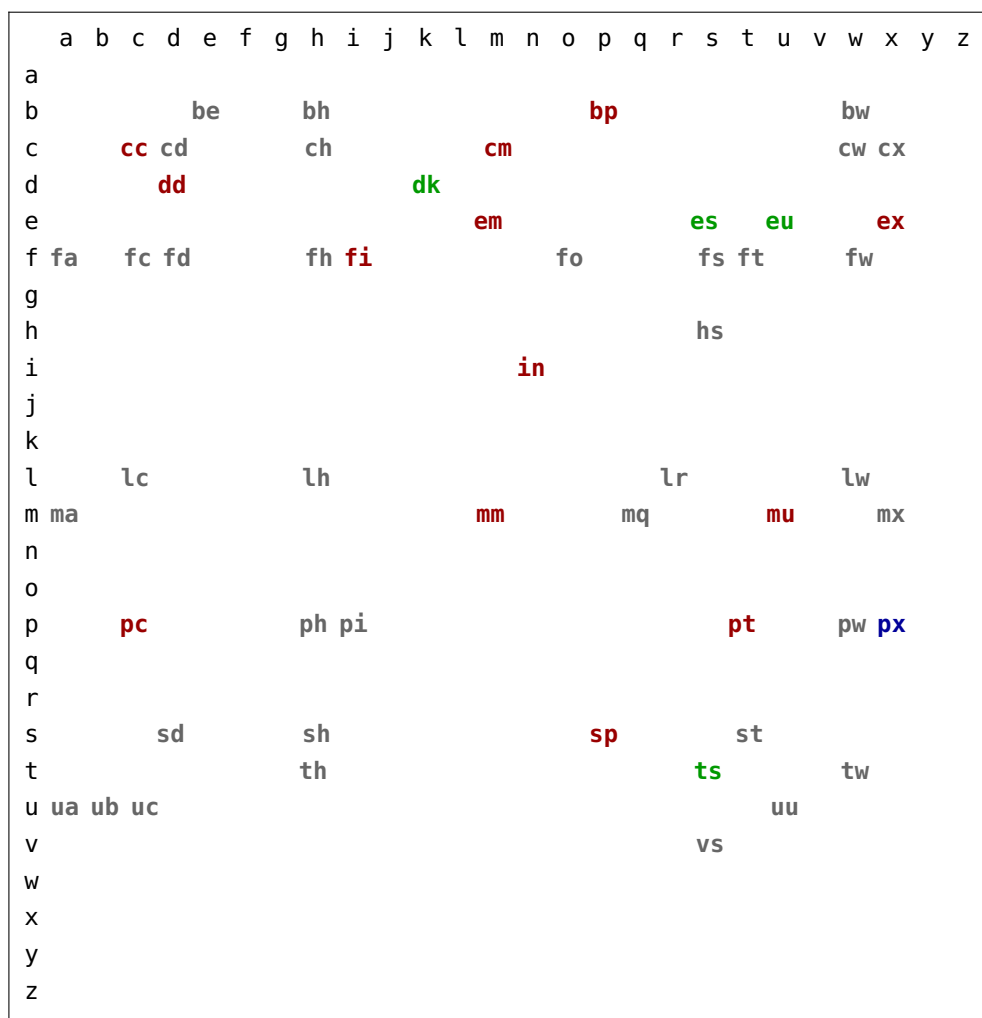


Figure 5.1 A map of available units

```
\startcombination[nx=4,ny=1]
  {\ruledhbox to 1cx{\strut one}} {1}
  {\ruledhbox to 1cx{\strut two}} {2}
  {\ruledhbox to 1cx{\strut three}} {3}
  {\ruledhbox to 1cx{\strut four}} {4}
\stopcombination
```

one.....	two.....	three.....	four.....
1	2	3	4

The uu can be set by users using the \uunit dimension variable. The default value is 1cm. Its current value is also known at the MetaPost end, as demonstrated in figure 5.2.

```
\startcombination[nx=2,ny=1]
```

```

\startcontent
  \uunit=1cm
  \framed[offset=1uu]
    \bgroup
      \startMPcode
        fill fullcircle scaled 3uu withcolor "darkred" ;
        fill fullcircle scaled 2cm withcolor "darkgreen" ;
      \stopMPcode
    \egroup
\stopcontent
\startcaption
  \type {\uunit = 1cm}
\stopcaption
\startcontent
  \uunit=1cx
  \framed[offset=.1uu]
    \bgroup
      \startMPcode
        fill fullcircle scaled .5uu withcolor "darkblue" ;
        fill fullcircle scaled 2cm withcolor "darkyellow" ;
      \stopMPcode
    \egroup
\stopcontent
\startcaption
  \type {\uunit = 1cx}
\stopcaption
\stopcombination

```

There is one catch here. If you use your own uu as numeric, you might need this:

```
save uu ; numeric uu ; uu := 1cm ;
```

That is: make sure the meaning is restored afterwards and explicitly declare the variable. But this is good practice anyway when you generate multiple graphics using the same MetaPost instance.

There are a few units not mentioned yet and those concern math, where we need to adapt to the current style.

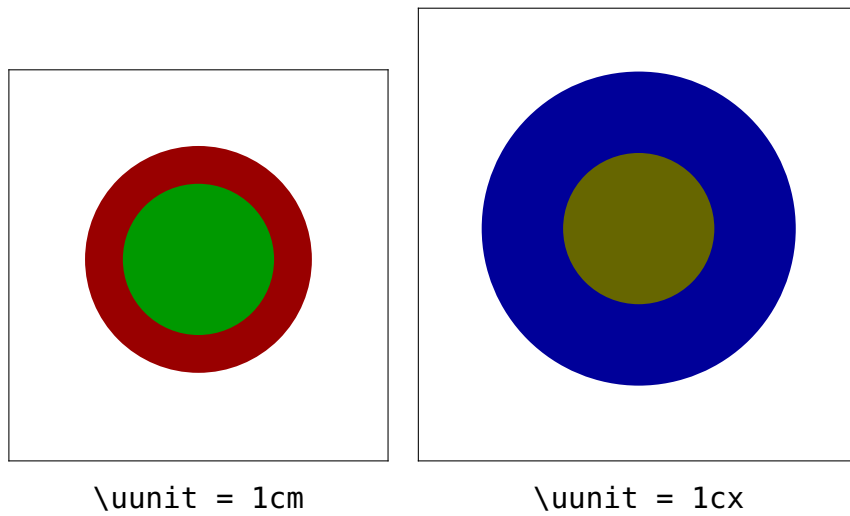
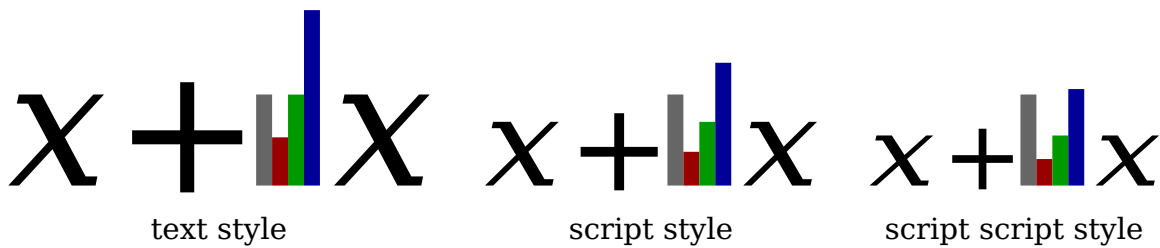


Figure 5.2 Shared user units in $\text{T}_{\text{E}}\text{X}$ and MetaFun.



The bars show 1ex, 1ma (axis), 1mx (ex-height) and 1mq (em-width or quad). The last three adapt themselves to the style. Often the mx makes more sense than ex.

5.6 Colofon

Author	Hans Hagen
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com contextgarden.net ntg-context@ntg.nl

6 Macros

The background of the entire image is a solid brown color. Overlaid on this background is a repeating pattern of thin, dark gray diagonal lines. These lines are slanted at approximately a 45-degree angle and are spaced evenly across the entire frame.

low level

TEX

macros

Contents

6.1	Preamble	101
6.2	Definitions	101
6.3	Runaway arguments	111
6.4	Introspection	112
6.5	nesting	113
6.6	Prefixes	116
6.7	Arguments	118
6.8	Constants	119
6.9	Passing parameters	120
6.10	Nesting	124
6.11	Duplicate hashes	125

6.1 Preamble

This chapter overlaps with other chapters but brings together some extensions to the macro definition and expansion parts. As these mechanisms were stepwise extended, the other chapters describe intermediate steps in the development.

Now, in spite of the extensions discussed here the main idea is still that we have \TeX act like before. We keep the charm of the macro language but these additions make for easier definitions, but (at least initially) none that could not be done before using more code.

6.2 Definitions

A macro definition normally looks like this:⁶

```
\def\macro#1#2%
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

Such a macro can be used as:

```
\macro {1}{2}
\macro {1} {2}  middle space gobbled
\macro 1 {2}    middle space gobbled
\macro {1} 2    middle space gobbled
```

⁶ The `\dontleavehmode` command make the examples stay on one line.

```
\macro 1 2      middle space gobbled
```

We show the result with some comments about how spaces are handled:

```
12|
12|      middle space gobbled
12|      middle space gobbled
12|      middle space gobbled
12|      middle space gobbled
```

A definition with delimited parameters looks like this:

```
\def\macro[#1]%
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\hss}}}
```

When we use this we get:

```
\macro [1]
\macro [ 1]      leading space kept
\macro [1 ]      trailing space kept
\macro [ 1 ]      both spaces kept
```

Again, watch the handling of spaces:

```
1|
1|      leading space kept
1|      trailing space kept
1|      both spaces kept
```

Just for the record we show a combination:

```
\def\macro[#1]#2%
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\type{#2}\vl\hss}}}
```

With this:

```
\macro [1]{2}
\macro [1] {2}
\macro [1] 2
```

we can again see the spaces go away:

```
12|
12|
```

```
|12|
```

A definition with two separately delimited parameters is given next:

```
\def\macro[#1#2]%
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

When used:

```
\macro [12]
\macro [ 12]    leading space gobbled
\macro [12 ]    trailing space kept
\macro [ 12 ]   leading space gobbled, trailing space kept
\macro [1 2]    middle space kept
\macro [ 1 2 ]  leading space gobbled, middle and trailing space kept
```

We get ourselves:

```
|12|
|12|    leading space gobbled
|12|    trailing space kept
|12|    leading space gobbled, trailing space kept
|1 2|   middle space kept
|1 2|   leading space gobbled, middle and trailing space kept
```

These examples demonstrate that the engine does some magic with spaces before (and therefore also between multiple) parameters.

We will now go a bit beyond what traditional T_EX engines do and enter the domain of LuaMetaT_EX specific parameter specifiers. We start with one that deals with this hard coded space behavior:

```
\def\macro[#^#^]%
  {\dontleavehmode\hbox to 6em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

The #[^] specifier will count the parameter, so here we expect again two arguments but the space is kept when parsing for them.

```
\macro [12]
\macro [ 12]
\macro [12 ]
\macro [ 12 ]
\macro [1 2]
```

```
\macro [ 1 2 ]
```

Now keep in mind that we could deal well with all kind of parameter handling in Con-TeXt for decades, so this is not really something we missed, but it complements the to be discussed other ones and it makes sense to have that level of control. Also, availability triggers usage. Nevertheless, some day the `#^` specifier will come in handy.

```
12|
12|
12|
12|
1 2|
1 2|
```

We now come back to an earlier example:

```
\def\macro[#1]%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\hss}}
```

When we use this we see that the braces in the second call are removed:

```
\macro [1]
\macro [{1}]
```

```
1| 1|
```

This can be prohibited by the `#+` specifier, as in:

```
\def\macro[#+]%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\hss}}
```

As we see, the braces are kept:

```
\macro [1]
\macro [{1}]
```

Again, we could easily get around that (for sure intended) side effect but it just makes nicer code when we have a feature like this.

```
1| {1}|
```

Sometimes you want to grab an argument but are not interested in the results. For this we have two specifiers: one that just ignores the argument, and another one that keeps counting but discards it, i.e. the related parameter is empty.

```
\def\macro[#1][#0][#3][#-][#4]%
  {\dontleavehmode\hbox spread 1em
    {\vl\type{#1}\vl\type{#2}\vl\type{#3}\vl\type{#4}\vl\hss}}}
```

The second argument is empty and the fourth argument is simply ignored which is why we need #4 for the fifth entry.

```
\macro [1][2][3][4][5]
```

Here is proof that it works:

```
|1|3|5|
```

The reasoning behind dropping arguments is that for some cases we get around the nine argument limitation, but more important is that we don't construct token lists that are not used, which is more memory (and maybe even cpu cache) friendly.

Spaces are always kind of special in T_EX, so it will be no surprise that we have another specifier that relates to spaces.

```
\def\macro[#1]#* [#2]%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}}
```

This permits usage like the following:

```
\macro [1][2]
\macro [1] [2]
```

```
|1|2| |1|2|
```

Without the optional 'grab spaces' specifier the second line would possibly throw an error. This because T_EX then tries to match][so the] [in the input is simply added to the first argument and the next occurrence of][will be used. That one can be someplace further in your source and if not T_EX complains about a premature end of file. But, with the #* option it works out okay (unless of course you don't have that second argument [2]).

Now, you might wonder if there is a way to deal with that second delimited argument being optional and of course that can be programmed quite well in traditional macro code. In fact, ConT_EXt does that a lot because it is set up as a parameter driven system with optional arguments. That subsystem has been optimized to the max over years and it works quite well and performance wise there is very little to gain. However, as soon as you enable tracing you end up in an avalanche of expansions and that is no fun.

This time the solution is not in some special specifier but in the way a macro gets defined.

```
\tolerant\def\macro[#1]*[#2]%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

The magic `\tolerant` prefix with delimited arguments and just quits when there is no match. So, this is acceptable:

```
\macro [1][2]
\macro [1] [2]
\macro [1]
\macro
```

```
|12| |12| |1| |
```

We can check how many arguments have been processed with a dedicated conditional:

```
\tolerant\def\macro[#1]*[#2]%
  {\ifarguments 0\or 1\or 2\or ?\fi: \vl\type{#1}\vl\type{#2}\vl}
```

We use this test:

```
\macro [1][2] \macro [1] [2] \macro [1] \macro
```

The result is: 2: |12| 2: |12| 1: |1| 0: | which is what we expect because we flush inline and there is no change of mode. When the following definition is used in display mode, the leading `n=` can for instance start a new paragraph and when code in `\everypar` you can loose the right number when macros get expanded before the `n` gets injected.

```
\tolerant\def\macro[#1]*[#2]%
  {n=\ifarguments 0\or 1\or 2\or ?\fi: \vl\type{#1}\vl\type{#2}\vl}
```

In addition to the `\ifarguments` test primitive there is also a related internal counter `\lastarguments` set that you can consult, so the `\ifarguments` is actually just a shortcut for `\ifcase\lastarguments`.

We now continue with the argument specifiers and the next two relate to this optional grabbing. Consider the next definition:

```
\tolerant\def\macro#1#*#2%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

With this test:

```
\macro {1} {2}
\macro {1}
\macro
```

We get:

```
|12| |1\macro|
```

This is okay because the last `\macro` is a valid (single token) argument. But, we can make the braces mandate:

```
\tolerant\def\macro#=#*#=#%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

Here the `#=` forces a check for braces, so:

```
\macro {1} {2}
\macro {1}
\macro
```

gives this:

```
|12| |1| |
```

However, we do loose these braces and sometimes you don't want that. Of course when you pass the results downstream to another macro you can always add them, but it was cheap to add a related specifier:

```
\tolerant\def\macro#_#*#_#%
  {\dontleavehmode\hbox spread 1em{\vl\type{#1}\vl\type{#2}\vl\hss}}
```

Again, the magic `\tolerant` prefix works will quit scanning when there is no match. So:

```
\macro {1} {2}
\macro {1}
\macro
```

leads to:

```
|{1}|{2}| |{1}| |
```

When you're tolerant it can be that you still want to pick up some argument later on. This is why we have a continuation option.

```

\tolerant\def\foo      [#1]*[#2]#:#3{!#1!#2!#3!}
\tolerant\def\oof[#1]*[#2]#:(#3)#:#4{!#1!#2!#3!#4!}
\tolerant\def\ofo      [#1]#:(#2)#:#3{!#1!#2!#3!}

```

Hopefully the next example demonstrates how it works:

```

\foo{3} \foo[1]{3} \foo[1][2]{3}
\oof{4} \oof[1]{4} \oof[1][2]{4}
\oof[1][2](3){4} \oof[1](3){4} \oof(3){4}
\ofo{3} \ofo[1]{3}
\ofo[1](2){3} \ofo(2){3}

```

As you can see we can have multiple continuations using the #: directive:

```

!!!3! !1!!!3! !1!2!3!
!!!!4! !1!!!!4! !1!2!!4!
!1!2!3!4! !1!!!3!4! !!!3!4!
!!!3! !1!!!3!
!1!2!3! !2!3!

```

The last specifier doesn't work well with the \ifarguments state because we no longer know what arguments were skipped. This is why we have another test for arguments. A zero value means that the next token is not a parameter reference, a value of one means that a parameter has been set and a value of two signals an empty parameter. So, it reports the state of the given parameter as a kind of \ifcase.

```

\def\foo#1#2{ [\ifparameter#1\or(ONE)\fi\ifparameter#2\or(TWO)\fi] }

```

Of course the test has to be followed by a valid parameter specifier:

```

\foo{1}{2} \foo{1}{} \foo{}{2} \foo{}{}

```

The previous code gives this:

```

[(ONE)(TWO)] [(ONE)] [(TWO)] []

```

A combination check \ifparameters, again a case, matches the first parameter that has a value set.

We could add plenty of specifiers but we need to keep in mind that we're not talking of an expression scanner. We need to keep performance in mind, so nesting and backtracking are no option. We also have a limited set of useable single characters, but here's one that uses a symbol that we had left:

```
\def\startfoo[#/#]/\stopfoo{ [#1](#2) }
```

The slash directive removes leading and trailing so called spacers as well as tokens that represent a paragraph end:

```
\startfoo [x ] x \stopfoo
\startfoo [ x ] x \stopfoo
\startfoo [ x] x \stopfoo
\startfoo [ x] \par x \par \par \stopfoo
```

So we get this:

```
[x](x) [x](x) [x](x) [x](x)
```

The next directive, the quitter #;, is demonstrated with an example. When no match has occurred, scanning picks up after this signal, otherwise we just quit.

```
\tolerant\def\foo[#1]#;(#2){/#1/#2/}
```

```
\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo(1)\quad\foo(2)\quad\foo(3)\par
```

```
\tolerant\def\foo[#1]#;#={/#1/#2/}
```

```
\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo{1}\quad\foo{2}\quad\foo{3}\par
```

```
\tolerant\def\foo[#1]#;#2{/#1/#2/}
```

```
\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo{1}\quad\foo{2}\quad\foo{3}\par
```

```
\tolerant\def\foo[#1]#;(#2)#;#={/#1/#2/#3/}
```

```
\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo(1)\quad\foo(2)\quad\foo(3)\par
\foo{1}\quad\foo{2}\quad\foo{3}\par
```

```
/1// /2// /3//
```

```
//1/ //2/ //3/
```

```
/1// /2// /3//
```

```
//1/ //2/ //3/
```

```
/1// /2// /3//
```

```
//1/ //2/ //3/
```

```
/1/// /2/// /3///
//1// //2// //3//
///1/ ///2/ ///3/
```

I have to admit that I don't really need it but it made some macros that I was redefining behave better, so there is some self-interest here. Anyway, I considered some other features, like picking up a detokenized argument but I don't expect that to be of much use. In the meantime we ran out of reasonable characters, but some day #? and #! might show up, or maybe I find a use for #< and #>. A summary of all this is given here:

+	keep the braces
-	discard and don't count the argument
/	remove leading and trailing spaces and pars
=	braces are mandate
_	braces are mandate and kept
^	keep leading spaces
1-9	an argument
0	discard but count the argument
*	ignore spaces
:	pick up scanning here
;	quit scanning
.	ignore pars and spaces
,	push back space when quit

The last two have not been discussed and were added later. The period directive gobbles space and par tokens and discards them in the process. The comma directive is like * but it pushes back a space when the matching quits.

```
\tolerant\def\foo[#1]#;(#2){/#1/#2/}
```

```
\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo(1)\quad\foo(2)\quad\foo(3)\par
```

```
\tolerant\def\foo[#1]#;#={/#1/#2/}
```

```
\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo{1}\quad\foo{2}\quad\foo{3}\par
```

```
\tolerant\def\foo[#1]#;#2{/#1/#2/}
```

```
\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo{1}\quad\foo{2}\quad\foo{3}\par
```

```
\tolerant\def\foo[#1]#;(#2)#;#={/#1/#2/#3/}
```

```
\foo[1]\quad\foo[2]\quad\foo[3]\par
\foo(1)\quad\foo(2)\quad\foo(3)\par
\foo{1}\quad\foo{2}\quad\foo{3}\par
```

```
/1// /2// /3//
//1/ //2/ //3/
/1// /2// /3//
//1/ //2/ //3/
/1// /2// /3//
//1/ //2/ //3/
/1/// /2/// /3///
//1// //2// //3//
///1/ ///2/ ///3/
```

Gobbling spaces versus pushing back is an interface design decision because it has to do with consistency.

6.3 Runaway arguments

There is a particular troublesome case left: a runaway argument. The solution is not pretty but it's the only way: we need to tell the parser that it can quit.

```
\tolerant\def\foo[#1=#2]%
  {\ifarguments 0\or 1\or 2\or 3\or 4\fi:\vl\type{#1}\vl\type{#2}\vl}
```

The outcome demonstrates that one still has to do some additional checking for sane results and there are alternative way to (ab)use this mechanism. It all boils down to a clever combination of delimiters and `\ignorearguments`.

```
\dontleavehmode \foo[a=1]
\dontleavehmode \foo[b=]
\dontleavehmode \foo[=]
\dontleavehmode \foo[x]\ignorearguments
```

All calls are accepted:

```
2:a|
2:b|
2:|
1:x|
```

Just in case you wonder about performance: don't expect miracles here. On the one hand there is some extra overhead in the engine (when defining macros as well as when collecting arguments during a macro call) and maybe using these new features can sort of compensate that. As mentioned: the gain is mostly in cleaner macro code and less clutter in tracing. And I just want the ConT_EXt code to look nice: that way users can look in the source to see what happens and not drown in all these show-off tricks, special characters like underscores, at signs, question marks and exclamation marks.

For the record: I normally run tests to see if there are performance side effects and as long as processing the test suite that has thousands of files of all kind doesn't take more time it's okay. Actually, there is a little gain in ConT_EXt but that is to be expected, but I bet users won't notice it, because it's easily offset by some inefficient styling. Of course another gain of loosing some indirectness is that error messages point to the macro that the user called for and not to some follow up.

6.4 Introspection

A macro has a meaning. You can serialize that meaning as follows:

```
\tolerant\protected\def\foo#1[#2]#* [#3]%
  {(1=#1) (2=#3) (3=#3)}

\meaning\foo
```

The meaning of \foo comes out as:

```
tolerant protected macro:#1[#2]#* [#3]->(1=#1) (2=#3) (3=#3)
```

When you load the module system-tokens you can also say:

```
\luatokenable\foo
```

This produces a table of tokens specifications:

```
tolerant protected macro:#1[#2]#* [#3]->(1=#1) (2=#3) (3=#3)
```

tolerant protected control sequence: foo					
596513	19	49	match		argument 1
596468	12	91	other char	[U+0005B	
592447	19	50	match		argument 2
592943	12	93	other char] U+0005D	
596633	19	42	match		argument *

593519	12	91	other char	[U+0005B	argument 3
596348	19	51	match			
596971	12	93	other char]	U+0005D	
594557	20	0	end match			
<hr/>						
591539	12	40	other char	(U+00028	
595427	12	49	other char	1	U+00031	
594487	12	61	other char	=	U+0003D	
595502	21	1	parameter reference			
593015	12	41	other char)	U+00029	
587553	10	32	spacer			
594033	12	40	other char	(U+00028	
593065	12	50	other char	2	U+00032	
593453	12	61	other char	=	U+0003D	
596704	21	3	parameter reference			
596563	12	41	other char)	U+00029	
596812	10	32	spacer			
587522	12	40	other char	(U+00028	
596701	12	51	other char	3	U+00033	
594568	12	61	other char	=	U+0003D	
594634	21	3	parameter reference			
595466	12	41	other char)	U+00029	

A token list is a linked list of tokens. The magic numbers in the first column are the token memory pointers. and because macros (and token lists) get recycled at some point the available tokens get scattered, which is reflected in the order of these numbers. Normally macros defined in the macro package are more sequential because they stay around from the start. The second and third row show the so called command code and the specifier. The command code groups primitives in categories, the specifier is an indicator of what specific action will follow, a register number a reference, etc. Users don't need to know these details. This macro is a special version of the online variant:

```
\showluatokens\foo
```

That one is always available and shows a similar list on the console. Again, users normally don't want to know such details.

6.5 nesting

You can nest macros, as in:

```
\def\foo#1#2{\def\oof##1{<#1>##1<#2>}}
```

At first sight the duplication of `#` looks strange but this is what happens. When `TEX` scans the definition of `\foo` it sees two arguments. Their specification ends up in the preamble that defines the matching. When the body is scanned, the `#1` and `#2` are turned into a parameter reference. In order to make nested macros with arguments possible a `#` followed by another `#` becomes just one `#`. Keep in mind that the definition of `\oof` is delayed till the macro `\foo` gets expanded. That definition is just stored and the only thing that get's replaced are the two references to a macro parameter

control sequence: foo

596653	19	49	match		argument 1
596608	19	50	match		argument 2
595439	20	0	end match		
596651	128	1	def		def
478946	146	0	tolerant call		oof
595340	6	35	parameter		
596825	12	49	other char	1	U+00031
594156	1	123	left brace		
594093	12	60	other char	<	U+0003C
596516	21	1	parameter reference		
587476	12	62	other char	>	U+0003E
596675	6	35	parameter		
596591	12	49	other char	1	U+00031
596851	12	60	other char	<	U+0003C
596810	21	2	parameter reference		
595502	12	62	other char	>	U+0003E
596644	2	125	right brace		

Now, when we look at these details, it might become clear why for instance we have ‘variable’ names like `#4` and not `#whatever` (with or without hash). Macros are essentially token lists and token lists can be seen as a sequence of numbers. This is not that different from other programming environments. When you run into buzzwords like ‘bytecode’ and ‘virtual machines’ there is actually nothing special about it: some high level programming (using whatever concept, and in the case of `TEX` it's macros) eventually ends up as a sequence of instructions, say bytecodes. Then you need some machinery to run over that and act upon those numbers. It's something you arrive at naturally when you play with interpreting languages.⁷

⁷ I actually did when I wrote an interpreter for some computer assisted learning system, think of a kind of interpreted Pascal, but later realized that it was a a bytecode plus virtual machine thing. I'd just applied what I learned when playing with eight bit processors that took bytes, and interpreted opcodes and such.

So, internally a #4 is just one token, a operator-operand combination where the operator is “grab a parameter” and the operand tells “where to store” it. Using names is of course an option but then one has to do more parsing and turn the name into a number⁸, add additional checking in the macro body, figure out some way to retain the name for the purpose of reporting (which then uses more token memory or strings). It is simply not worth the trouble, let alone the fact that we loose performance, and when T_EX showed up those things really mattered.

It is also important to realize that a # becomes either a preamble token (grab an argument) or a reference token (inject the passed tokens into a new input level). Therefore the duplication of hash tokens ## that you see in macro nested bodies also makes sense: it makes it possible for the parser to distinguish between levels. Take:

```
\def\foo#1{\def\oof##1{#1##1#1}}
```

Of course one can think of this:

```
\def\foo#fence{\def\oof#text{#fence#text#fence}}
```

But such names really have to be unique then! Actually ConT_EXt does have an input method that supports such names, but discussing it here is a bit out of scope. Now, imagine that in the above case we use this:

```
\def\foo[#1][#2]{\def\oof##1{#1##1#2}}
```

If you're a bit familiar with the fact that T_EX has a model of category codes you can imagine that a predictable “hash followed by a number” is way more robust than enforcing the user to ensure that catcodes of ‘names’ are in the right category (read: is a bracket part of the name or not). So, say that we go completely arbitrary names, we then suddenly needs some escaping, like:

```
\def\foo[#{left}][#{right}]{\def\oof#{text}{#{left}#{text}#{right}}}
```

And, if you ever looked into macro packages, you will notice that they differ in the way they assign category codes. Asking users to take that into account when defining macros makes not that much sense.

So, before one complains about T_EX being obscure (the hash thing), think twice. Your demand for simplicity for your coding demand will make coding more cumbersome for

There's nothing spectacular about all this and I only realized decades later that the buzzwords describes old natural concepts.

⁸ This is kind of what MetaPost does with parameters to macros. The side effect is that in reporting you get text0, expr2 and such reported which doesn't make things more clear.

the complex cases that macro packages have to deal with. It's comparable using T_EX for input or using (say) mark down. For simple documents the later is fine, but when things become complex, you end up with similar complexity (or even worse because you lost the enforced detailed structure). So, just accept the unavoidable: any language has its peculiar properties (and for sure I do know why I dislike some languages for it). The T_EX system is not the only one where dollars, percent signs, ampersands and hashes have special meaning.

6.6 Prefixes

Traditional T_EX has three prefixes that can be used with macros: `\global`, `\outer` and `\long`. The last two are no-op's in LuaMetaT_EX and if you want to know what they do (did) you can look it up in the T_EXbook. The ε -T_EX extension gave us `\protected`.

In LuaMetaT_EX we have `\global`, `\protected`, `\tolerant` and overload related prefixes like `\frozen`. A protected macro is one that doesn't expand in an expandable context, so for instance inside an `\edef`. You can force expansion by using the `\expand` primitive in front which is also something LuaMetaT_EX.

Frozen macros cannot be redefined without some effort. This feature can to some extent be used to prevent a user from overloading, but it also makes it harder for the macro package itself to redefine on the fly. You can remove the lock with `\unletfrozen` and add a lock with `\letfrozen` so in the end users still have all the freedoms that T_EX normally provides.

```

\def\foo{foo} 1: \meaning\foo
\frozen\def\foo{foo} 2: \meaning\foo
\unletfrozen \foo 3: \meaning\foo
\protected\def\foo{foo} 4: \meaning\foo
\unletfrozen \foo 5: \meaning\foo

```

```

1: macro:foo
2: macro:foo
3: macro:foo
4: protected macro:foo
5: protected macro:foo

```

This actually only works when you have set `\overloadmode` to a value that permits redefining a frozen macro, so for the purpose of this example we set it to zero.

A `\tolerant` macro is one that will quit scanning arguments when a delimiter cannot be matched. We saw examples of that in a previous section.

These prefixes can be chained (in arbitrary order):

```
\frozen\tolerant\protected\global\def\foo[#1]#*[#2]{...}
```

There is actually an additional prefix, `\immediate` but that one is there as signal for a macro that is defined in and handled by Lua. This prefix can then perform the same function as the one in traditional \TeX , where it is used for backend related tasks like `\write`.

Now, the question is of course, to what extent will $\text{Con}\text{\TeX}t$ use these new features. One important argument in favor of using `\tolerant` is that it gives (hopefully) better error messages. It also needs less code due to lack of indirectness. Using `\frozen` adds some safeguards although in some places where $\text{Con}\text{\TeX}t$ itself overloads commands, we need to defrost. Adapting the code is a tedious process and it can introduce errors due to mistypings, although these can easily be fixed. So, it will be used but it will take a while to adapt the code base.

One problem with frozen macros is that they don't play nice with for instance `\futurelet`. Also, there are places in $\text{Con}\text{\TeX}t$ where we actually do redefine some core macro that we also want to protect from redefinition by a user. One can of course `\unletfrozen` such a command first but as a bonus we have a prefix `\overloaded` that can be used as prefix. So, one can easily redefine a frozen macro but it takes a little effort. After all, this feature is mainly meant to protect a user for side effects of definitions, and not as final blocker.⁹

A frozen macro can still be overloaded, so what if we want to prevent that? For this we have the `\permanent` prefix. Internally we also create primitives but we don't have a prefix for that. But we do have one for a very special case which we demonstrate with an example:

```
\def\F00 % trickery needed to pick up an optional argument
  {\noalign{\vskip10pt}}

\noaligned\protected\tolerant\def\00F[#1]%
  {\noalign{\vskip\iftok{#1}\emptytoks10pt\else#1\fi}}

\starttabulate[|l|l|]
  \NC test \NC test \NC \NR
  \NC test \NC test \NC \NR
```

⁹ As usual adding features like this takes some experimenting and we're now at the third variant of the implementation, so we're getting there. The fact that we can apply such features in large macro package like $\text{Con}\text{\TeX}t$ helps figuring out the needs and best approaches.

```

\F00
\NC test \NC test \NC \NR
\00F[30pt]
\NC test \NC test \NC \NR
\00F
\NC test \NC test \NC \NR
\stoptabulate

```

When \TeX scans input (from a file or token list) and starts an alignment, it will pick up rows. When a row is finished it will look ahead for a `\noalign` and it expands the next token. However, when that token is protected, the scanner will not see a `\noalign` in that macro so it will likely start complaining when that next macro does get expanded and produces a `\noalign` when a cell is built. The `\noaligned` prefix flags a macro as being one that will do some `\noalign` as part of its expansion. This trick permits clean macros that pick up arguments. Of course it can be done with traditional means but this whole exercise is about making the code look nice.

The table comes out as:

```

test  test
test  test

test  test

```

```

test  test

```

```

test  test

```

One can check the flags with `\iff flags` which takes a control sequence and a number, where valid numbers are:

1 frozen	2 permanent	4 immutable	8 primitive
16 mutable	32 noaligned	64 instance	

The level of checking is controlled with the `\overloadmode` but I'm still not sure about how many levels we need there. A zero value disables checking, the values 1 and 3 give warnings and the values 2 and 4 trigger an error.

6.7 Arguments

The number of arguments that a macro takes is traditionally limited to nine (or ten if one takes the trailing `#` into account). That this is enough for most cases is demonstrated

by the fact that ConT_EXt has only a handful of macros that use #9. The reason for this limitation is in part a side effect of the way the macro preamble and arguments are parsed. However, because in LuaMetaT_EX we use a different implementation, it was not that hard to permit a few more arguments, which is why we support upto 15 arguments, as in:

```
\def\foo#1#2#3#4#5#6#7#8#9#A#B#C#D#E#F{...}
```

We can support the whole alphabet without much trouble but somehow sticking to the hexadecimal numbers makes sense. It is unlikely that the core of ConT_EXt will use this option but sometimes at the user level it can be handy. The penalty in terms of performance can be neglected.

```
\tolerant\def\foo#=#=#=#=#=#=#=#=#=#=#=#=#=#=#=#%
{(#1)(#2)(#3)(#4)(#5)(#6)(#7)(#8)(#9)(#A)(#B)(#C)(#D)(#E)(#F)}

\foo{1}{2}
```

In the previous example we have 15 optional arguments where braces are mandate (otherwise we the scanner happily scoops up what follows which for sure gives some error).

6.8 Constants

The LuaMetaT_EX engine has lots of efficiency tricks in the macro parsing and expansion code that makes it not only fast but also let is use less memory. However, every time that the body of a macro is to be injected the expansion machinery kicks in. This often means that a copy is made (pushed in the input and used afterwards). There are however cases where the body is just a list of character tokens (with category letter or other) and no expansion run over the list is needed.

It is tempting to introduce a string data type that just stores strings and although that might happen at some point it has the disadvantage that one need to tokenize that string in order to be able to use it, which then defeats the gain. An alternative has been found in constant macros, that is: a macro without parameters and a body that is considered to be expanded and never freed by redefinition. There are two variants:

```
\cdef      \foo      {whatever}
\cdefcsname foo\endcsname{whatever}
```

These are actually just equivalents to

```
\edef      \foo      {whatever}
```

```
\edefcsname foo\endcsname{whatever}
```

just to make sure that the body gets expanded at definition time but they are also marked as being constant which in some cases might give some gain, for instance when used in csname construction. The gain is less then one expects although there are a few cases in ConT_EXt where extreme usage of parameters benefits from it. Users are unlikely to use these two primitives.

Another example of a constant usage is this:

```
\lettonothing\foo
```

which gives \foo an empty body. That one is used in the core, if only because it gives a bit smaller code. Performance is no that different from

```
\let\foo\empty
```

but it saves one token (8 bytes) when used in a macro. The assignment itself is not that different because \foo is made an alias to \empty which in turn only needs incrementing a reference counter.

6.9 Passing parameters

When you define a macro, the #1 and more parameters are embedded as a reference to a parameter that is passed. When we have four parameters, the parameter stack has four entries and when an entry is eventually accessed a new input level is pushed and tokens are fetched from that list. This has some side effects when we check a parameter. This can happen multiple times, depending on how often we access a parameter. Take the following:

```
\def\oof#1{#1}
```

```
\tolerant\def\foo[#1]#*[#2]%
  {1:\ifparameter#1\or Y\else N\fi\quad
  2:\ifparameter#2\or Y\else N\fi\quad
  \oof{3:\ifparameter #1\or Y\else N\fi\quad
    4:\ifparameter #2\or Y\else N\fi\quad}%
  \par}
```

```
\foo \foo[] \foo[][] \foo[A] \foo[A][B]
```

This gives:

```

1:N 2:N 3:N 4:N
1:N 2:N 3:N 4:N
1:N 2:N 3:N 4:N
1:Y 2:N 3:Y 4:N
1:Y 2:Y 3:Y 4:Y

```

as you probably expect. However the first two checks are different from the embedded checks because they can check against the parameter reference. When we expand `\oof` its argument gets passed to the macro as a list and when the scanner collects the next token it will then push the parameter content on the input stack. So, then, instead of a reference we get the referenced parameter list. Internally that means that in 3 and 4 we check for a token and not for the length of the list (as in case 1 & 2). This means that

```

\iftok{#1}\emptytoks Y\else N\fi
\ifparameter#1\or Y\else N\fi

```

are different. In the first case we have a proper token list and nested conditionals in that list are okay. In the second case we just look ahead to see if there is an `\or`, `\else` or other condition related command and if so we decide that there is no parameter. So, if `\ifparameter` is a suitable check for empty depends on the need for expansion.

When you define macros that themselves call macros that should operate on the arguments of its parent you can easily pass these:

```

\def\foo#1#2%
  {\oof{#1}{#2}{P}%
   \oof{#1}{#2}{Q}%
   \oof{#1}{#2}{R}}

\def\oof#1#2#3%
  {[#1][#1]%
   #3%
   [#2][#2]}

```

Here the nested call to `\oof` involved three passed parameters. You can avoid that as follows:

```

\def\foo#1#2%
  {\def\MyIndexOne{#1}%
   \def\MyIndexTwo{#2}%
   \oof{P}\oof{Q}\oof{R}}

```

```
\def\oof#1%
  {(\MyIndexOne)(\MyIndexOne)%
   #1%
   (\MyIndexTwo)(\MyIndexTwo)}
```

You can also do this:

```
\def\foo#1#2%
  {\def\oof##1%
   {/#1/#2/%
    ##1%
    /#1//#2/}%
   \oof{P}\oof{Q}\oof{R}}
```

These parameters indicated by # in the macro body are in fact references. When we call for instance `\foo{1}{2}` the two parameters get pushed on a parameter stack and the embodied references point to these stack entries. By the time that body gets expanded \TeX bumps the input level and pushes the parameter list onto the input stack. It then continues expansion. The parameter is not copied, because it can't be changed anyway. The only penalty in terms of performance and memory usage is the pushing and popping of the input. So how does that work out for these three cases?

When in the first case the `\oof{#1}{#2}{P}` is seen, \TeX starts expanding the `\oof` macro. That one expects three arguments. The `#1` reference is seen and in this case a copy of that parameter is passed. The same is true for the other two. Then, inside `\oof` expansion happens on the parameters on the stack and no copies have to be made there.

The second case defines two macros so again two copies are made that make the bodies of these macros. This comes at the cost of some runtime and memory. However, this time with `\oof{P}` only one argument gets passed and instead expansion of the macros happen in there.

Normally macro arguments are not that large but there can be situations where we really want to avoid useless copying. This not only saves memory but also can give a bit better performance. In the examples above the second variant is some 10% faster than the first one. We can gain another 10% with the following trick:

```
\def\foo#1#2%
  {\parameterdef\MyIndexOne\plusone % 1
   \parameterdef\MyIndexTwo\plustwo % 2
   \oof{P}\oof{Q}\oof{R}\norelax}
```

```
\def\oof#1%
  {\<\MyIndexOne><\MyIndexOne>%
   #1%
   <\MyIndexTwo><\MyIndexTwo>}
```

Here we define an explicit parameter reference that we access later on. There is the overhead of a definition but it can be neglected. We use that reference (abstraction) in \oof. Actually you can use that reference in any call down the chain.

When applied to \foo{1}{2} the four variants above give us:

```
[1][1]P[2][2][1][1]Q[2][2][1][1]R[2][2]
(1)(1)P(2)(2)(1)(1)Q(2)(2)(1)(1)R(2)(2)
/1/2/P/1//2//1/2/Q/1//2//1/2/R/1//2/
<1><1>P<2><2><1><1>Q<2><2><1><1>R<2><2>
```

Before we had parameterdef we had this:

```
\def\foo#1#2%
  {\integerdef\MyIndexOne\parameterindex\plusone % 1
   \integerdef\MyIndexTwo\parameterindex\plustwo % 2
   \oof{P}\oof{Q}\oof{R}\norelax}

\def\oof#1%
  {\<\expandparameter\MyIndexOne><\expandparameter\MyIndexOne>%
   #1%
   <\expandparameter\MyIndexTwo><\expandparameter\MyIndexTwo>}
```

It involves more tokens, is a bit less abstract, but as it is a cheap extension we kept it. It actually demonstrates that one can access parameters in the stack by index, but it one then needs to keep track of where access takes place. In principle one can debug the call chain this way.

To come back to performance and memory usage, when the arguments become larger the fourth variant with the \parameterdef quickly gains over the others. But it only shows in exceptional usage. This mechanism is more about abstraction: it permits us to efficiently turn arguments into local variables without the overhead involved in creating macros. You can test if a parameter is set

```
\tolerant\protected\def\MyMacro[#1]#:#2%
  {\parameterdef\MyArgumentOne\plusone
   \parameterdef\MyArgumentTwo\plustwo
   \ifparameter\MyArgumentOne\or
```

```
(\MyArgumentOne)
\fi
/\MyArgumentTwo/}
```

```
\MyMacro[one]{two}
\MyMacro{two}
```

Indeed we get:

```
(one) /two/ /two/
```

Of course `\ifparameter#1\or...` is more efficient but once you use named parameters like this it's probably not something you're worry too much about,

6.10 Nesting

We also have a few preamble features that relate to nesting. Although we can do without (as shown for years in LMTX) they do have some benefits. They are discussed as group here and because they are only useful for low level programming we stick to simple examples. The `#L` and `#R` use the following token as delimiters. Here we use `[` and `]` but they can be a `\cs` as well. Nested delimiters are handled well.

The `#S` grabs the argument till the next final square bracket `]` but in the process will grab nested with it sees a `[`. The `#P` does the same for parentheses and `#X` for angle brackets. In the next examples the `#*` just gobbles optional spaces but we've seen that one already.

The `#G` argument just registers the next token as delimiter but it will grab multiple of them. The `#M` gobbles more: in addition to the delimiter spaces are gobbled.

```
\tolerant\def\fooA      [#1]{(#1)}
\tolerant\def\fooB      [#L[#R]#1]{(#1)}
\tolerant\def\fooC      #S#1{(#1)}
\tolerant\def\fooE      #S#1,{(#1)}
\tolerant\def\fooF      #S#1#*#S#2{(#1/#2)}
\tolerant\def\fooG      [#1]#S[#2]#*#S[#3]{(#1/#2/#3)}
\tolerant\def\fooH      [#1][#S#2]#*[#S#3]{(#1/#2/#3)}
\tolerant\def\fooI      #1=#2#G,{(#1=#2)}
\tolerant\def\fooJ      #1=#2#M,{(#1=#2)}
```



```
\fooA[x]      (x)      (x)
\fooB[x]      (x)      (x)
```

<code>\fooC[1[2]3[4]5]</code>	<code>([1[2]3[4]5])</code>	<code>(1[2]3[4]5)</code>
<code>\fooE X[,]X,</code>	<code>(X[,]X)</code>	<code>(X[,]X)</code>
<code>\fooF[A] [B]</code>	<code>([A]/[B])</code>	<code>(A/B)</code>
<code>\fooF[] []</code>	<code>([]/[])</code>	<code>(/)</code>
<code>\fooG[a][b][c]</code>	<code>(a/b/c)</code>	<code>(a/b/c)</code>
<code>\fooG[a][b]</code>	<code>(a/b/)</code>	<code>(a/b/)</code>
<code>\fooG[a]</code>	<code>(a//)</code>	<code>(a//)</code>
<code>\fooG[a][x[x]x][c]</code>	<code>(a/x[x]x/c)</code>	<code>(a/x[x]x/c)</code>
<code>\fooH[a][x[x]x][c]</code>	<code>(a/x[x]x/c)</code>	<code>(a/x[x]x/c)</code>
<code>\fooI X=X,,,</code>	<code>(X=X)</code>	<code>(X=X)</code>
<code>\fooJ X=X, , ,</code>	<code>(X=X)</code>	<code>(X=X)</code>

These features make it possible to support nested setups more efficiently and also makes it possible to accept values that contain balanced brackets in setup commands without additional overhead. Although it has never been an issue to let users specify:

```
\defineoverlay[whatever][{some \command[withparameters] here}]
```

```
\setupfoo[before={\blank[big]}]
```

it might be less confusing to permit:

```
\defineoverlay[whatever][some \command[withparameters] here]
```

```
\setupfoo[before=\blank[big]]
```

as well, if only because occasionally users get hit by this.

6.11 Duplicate hashes

In \TeX every character has a so called category code. Most characters are classified as ‘letter’ (they make up words) or as ‘other’. In Unicode we distinguish symbols, punctuation, and more, but in \TeX these are all of category ‘other’. In math however we can classify them differently but in this perspective we ignore that. The backslash has category ‘escape’ and it starts a control sequence. The curly braces are (internally) of category ‘left brace’ and ‘right brace’ aka ‘begin group’ and ‘end group’ but, no matter what they are called, they begin and end something: a group, argument, token list, box, etc. Any character can have those categories. Although it would look strange to a \TeX user, this can be made valid:

```
!protected !gdef !weird¶1
B
```

```

    something: ¶1
E
!weird BhereE

```

In such a setup spaces can be of category ‘invisible’. The paragraph symbol takes the place of the hash as parameter identifier. The next code shows how this is done. Here we wrap all in a macro so that we don't get catcode interference in the document source.

```

\def\NotSoTeX
{
  \begingroup
  \catcode `B \begingroupcatcode
  \catcode `E \endgroupcatcode
  \catcode `¶ \parametercatcode
  \catcode `! \escapecatcode
  \catcode 32 \ignorecatcode
  \catcode 13 \ignorecatcode
  % this buffer has a definition:
  \getbuffer
  % which is now known globally
  \endgroup
}
\NotSoTeX
\weird{there}

```

This results in:

```

something:here
something:there

```

In the first line the !, B and E are used as escape and argument delimiters, in the second one we use the normal characters. When we show the `\meaningasis` we get:

```
\global \protected \def \weird #1{something:#1}
```

or in more detail:

protected control sequence: weird

595355	19	49	match		argument 1
592783	20	0	end match		
596378	11	115	letter	s	U+00073
596547	11	111	letter	o	U+0006F
597372	11	109	letter	m	U+0006D
595373	11	101	letter	e	U+00065

593866	11	116	letter	t	U+00074
596806	11	104	letter	h	U+00068
595236	11	105	letter	i	U+00069
596551	11	110	letter	n	U+0006E
596681	11	103	letter	g	U+00067
596635	12	58	other char	:	U+0003A
596513	21	1	parameter reference		

So, no matter how we set up the system, in the end we get some generic representation. When we see #1 in 'print' it can be either two tokens, # (catcode parameter) followed by 1 with catcode other, or one token referring to parameter 1 where the character 1 is the opcode of an internal 'reference command'. In order to distinguish a reference from the two token case, parameter hash tokens get shown as doubles.

```
\def\test #1{x#1x##1x####1x}
\def\tset ¶1{x¶1x¶¶1x¶¶¶¶1x}
```

And with \meaning we get, consistent with the input:

```
macro:#1->x#1x##1x####1x
macro:#1->x#1x¶¶1x¶¶¶¶1x
```

These are equivalent, apart from the parameter character in the body of the definition:

control sequence: test

592422	19	49	match		argument 1
597458	20	0	end match		

597380	11	120	letter	x	U+00078
596517	21	1	parameter reference		
594473	11	120	letter	x	U+00078
596053	6	35	parameter		
597831	12	49	other char	1	U+00031
596973	11	120	letter	x	U+00078
593761	6	35	parameter		
593058	6	35	parameter		
597453	12	49	other char	1	U+00031
595837	11	120	letter	x	U+00078

control sequence: tset

597582	19	49	match		argument 1
--------	----	----	-------	--	------------

596663	20	0	end match		
596850	11	120	letter	x	U+00078
593097	21	1	parameter reference		
595799	11	120	letter	x	U+00078
479238	6	182	parameter		
597583	12	49	other char	1	U+00031
596852	11	120	letter	x	U+00078
597418	6	182	parameter		
596976	6	182	parameter		
597159	12	49	other char	1	U+00031
597713	11	120	letter	x	U+00078

Watch how every ‘parameter’ is just a character with the Unicode index of the used input character as property. Let us summarize the process. When a single parameter character is seen in the input, the next character determines how it will be interpreted. If there is a digit then it becomes a reference to a parameter in the preamble, and when followed by another parameter character it will be appended to the body of the macro and that second one is dropped. So, two parameter characters become one, and four become two. One parameter character becomes a reference and from that you can guess what three in a row become. However, when $\text{T}_{\text{E}}\text{X}$ is showing the macro definition (using meaning) the hashes get duplicated in order to distinguish parameter references from parameter characters that were kept (e.g. for nested definitions). One can make an argument for `\parameterchar` as we also have `\escapechar` but by now this convention is settled and it doesn't look that bad anyway.

We now come to the more tricky part with respect to the doubling of hashes. When $\text{T}_{\text{E}}\text{X}$ was written its application landscape looked a bit different. For instance, fonts were limited and therefore it was natural to access special characters by name. Using `\#` to get a hash in the text was not that problematic, if one needed that character at all. The same can be said for the braces, backslash and even the dollar (after all $\text{T}_{\text{E}}\text{X}$ is free software).

But what if we have more visualization and/or serialization than meanings and tracing? When we opened up the internals in $\text{LuaT}_{\text{E}}\text{X}$ and even more in $\text{LuaMetaT}_{\text{E}}\text{X}$ the duplicating of hashes became a bit of a problem. There we don't need to distinguish between a parameter reference and a parameter character because by that time these references are resolved. All hashes that we encounter are just that: hashes. And this is why in $\text{LuaMetaT}_{\text{E}}\text{X}$ we disable the duplication for those cases where it serves no purpose.

When the engine scans a macro definition it starts with picking up the name of the macro. Then it starts scanning the preamble up to the left brace. In the preamble of a

Duplicate hashes

macro the scanner converts hashes followed by another token into single match token. Then when the macro body is scanned single hashes followed by a number become a reference, while double hashes become one hash and get interpreted at expansion time (possibly triggering an error when not followed by a valid specifier like a number). In traditional T_EX we basically had this:

```
\def\test#1{#1}
\def\test#1{##}
\def\test#1{#X}
\def\test#1{##1}
```

There can be a trailing # in the preamble for special purposes but we forget about that now. The first definition is valid, the second definition is invalid when the macro is expanded and the third definition triggers an error at definition time. The last definition will again trigger an error at expansion time.

However, in LuaMetaT_EX we have an extended preamble where the following preamble parameters are handled (some only in tolerant mode):

#n	parameter	index 1 upto E
#0	throw away parameter	increment index
#-	ignore parameter	keep index
#*	gobble white space	
#+	keep (honor) the braces	
#.	ignore pars and spaces	
#,	push back space when no match	
#/	remove leading and trailing spaces and pars	
#=	braces are mandate	
#^	keep leading spaces	
#_	braces are mandate and kept (obey)	
#@	par delimiter	only for internal usage
#:	pick up scanning here	
#;	quit scanning	
#L	left delimiter token	followed by token
#R	right delimiter token	followed by token
#G	gobble token	followed by token
#M	gobble token and spaces	followed by token

Duplicate hashes

#S	nest square brackets	only inner pairs
#X	nest angle brackets	only inner pairs
#P	nest parentheses	only inner pairs

As mentioned these will become so called match tokens and only when we show the meaning the hash will show up again.

```
\def\test[#1]#*[*S#2]{.#1.#2.}
```

control sequence: test

593064	12	91	other char	[U+0005B	
593649	19	49	match			argument 1
596701	12	93	other char]	U+0005D	
147444	19	42	match			argument *
597153	12	91	other char	[U+0005B	
595399	12	42	other char	*	U+0002A	
595794	11	83	letter	S	U+00053	
595445	19	50	match			argument 2
593162	12	93	other char]	U+0005D	
596907	20	0	end match			
<hr/>						
596781	12	46	other char	.	U+0002E	
595829	21	1	parameter reference			
597466	12	46	other char	.	U+0002E	
596738	21	2	parameter reference			
597861	12	46	other char	.	U+0002E	

This means that in the body of a macro you will not see `#*` show up. It is just a directive that tells the macro parser that spaces are to be skipped. The `#S` directive makes the parser for the second parameter handle nested square bracket. The only hash that we can see end up in the body is the one that we entered as double hash (then turned single) followed by (in traditional terms) a number that when all gets parsed with then become a reference: the sequence `##1` internally is `#1` and becomes ‘reference to parameter 1’ assuming that we define a macro in that body. If no number is there, an error is issued. This opens up the possibility to add more variants because it will only break compatibility with respect to what is seen as error. As with the preamble extensions, old documents that have them would have crashed before they became available.

So, this means that in the body, and actually anywhere in the document apart from preambles, we now support the following general parameter specifiers. Keep in mind that they expand in an expansion context which can be tricky when they overlap with

Duplicate hashes

preamble entries, like for instance `#R` in such an expansion. Future extensions can add more so *any* hashed shortcut is sensitive for that.

<code>#I</code>	current iterator	<code>\currentloopiterator</code>
<code>#P</code>	parent iterator	<code>\previousloopiterator 1</code>
<code>#G</code>	grandparent iterator	<code>\previousloopiterator 2</code>
<code>#H</code>	hash escape	<code>#</code>
<code>#S</code>	space escape	<code>␣</code>
<code>#T</code>	tab escape	<code>\t</code>
<code>#L</code>	newline escape	<code>\n</code>
<code>#R</code>	return escape	<code>\r</code>
<code>#X</code>	backslash escape	<code>\</code>
<code>#N</code>	nbsp	U+00A0 (under consideration)
<code>#Z</code>	zws	U+200B (under consideration)

Some will now argue that we already have `^^` escapes in $\text{T}_{\text{E}}\text{X}$ and `^^^^` and `^^^^^^` in $\text{LuaT}_{\text{E}}\text{X}$ and that is true. However, these can be disabled, and in $\text{ConT}_{\text{E}}\text{Xt}$ they are, where we instead enable the prescript, postscript, and index features in mathmode and there type `^` and `_` are used. Even more: in $\text{ConT}_{\text{E}}\text{Xt}$ we just let `^`, `_` and `&` be what they are. Occasionally I consider `$` to be just that but as I don't have dollars I will happily leave that for inline math. When users are not defining macros or are using the alternative definitions we can consider making the `#` a hash. An excellent discussion of how $\text{T}_{\text{E}}\text{X}$ reads it's input and changes state accordingly can be found in Victor Eijkhout's " $\text{T}_{\text{E}}\text{X}$ By Topic", section 2.6: when `^^` is followed by a character with $v < 128$ the interpreter will inject a character with code $v - 64$. When followed by two (!) lowercase hexadecimal characters, the corresponding character will be injected. Anyway, it not only looks kind of ugly, it also is somewhat weird because what follows is interpreted mixed way. The substitution happens early on (which is okay). But, how about the output? Traditional $\text{T}_{\text{E}}\text{X}$ serializes special characters with a similar syntax but that has become optional when eight bit mode was added to the engines, it is configurable in $\text{LuaT}_{\text{E}}\text{X}$ and has been dropped in $\text{LuaMetaT}_{\text{E}}\text{X}$: we operate in a utf universum.

6.11 Colofon

Author	Hans Hagen
$\text{ConT}_{\text{E}}\text{Xt}$	2025.02.12 08:44
$\text{LuaMetaT}_{\text{E}}\text{X}$	2.11.06 20250218
Support	www.pragma-ade.com contextgarden.net ntg-context@ntg.nl

7 Grouping



low level

TEX

grouping

Contents

7.1	Introduction	133
7.2	Kinds of grouping	135
7.3	Hooks	137
7.4	Local versus global	138
7.5	Files	141

7.1 Introduction

This is a rather short explanation. I decided to write it after presenting the other topics at the 2019 ConT_EXt meeting where there was a question about grouping.

7.1.1 Pascal

In a language like Pascal, the language that T_EX has been written in, or Modula, its successor, there is no concept of grouping like in T_EX. But we can find keywords that suggests this:

```
for i := 1 to 10 do begin ... end
```

This language probably inspired some of the syntax of T_EX and MetaPost. For instance an assignment in MetaPost uses `:=` too. However, the `begin` and `end` don't really group but define a block of statements. You can have local variables in a procedure or function but the block is just a way to pack a sequence of statements.

7.1.2 T_EX

In T_EX macros (or source code) the following can occur:

```
\begingroup
```

```
...
```

```
\endgroup
```

as well as:

```
\bgroup
```

```
...
```

```
\egroup
```

Here we really group in the sense that assignments to variables inside a group are forgotten afterwards. All assignments are local to the group unless they are explicitly done global:

```
\scratchcounter=1
\def\foo{foo}
\begingroup
  \scratchcounter=2
  \global\globalscratchcounter=2
  \gdef\foo{F00}
\endgroup
```

Here `\scratchcounter` is still one after the group is left but its global counterpart is now two. The `\foo` macro is also changed globally.

Although you can use both sets of commands to group, you cannot mix them, so this will trigger an error:

```
\bgroup
\endgroup
```

The bottomline is: if you want a value to persist after the group, you need to explicitly change its value globally. This makes a lot of sense in the perspective of $\text{T}_{\text{E}}\text{X}$.

7.1.3 MetaPost

The MetaPost language also has a concept of grouping but in this case it's more like a programming language.

```
begingroup ;
  n := 123 ;
endgroup ;
```

In this case the value of `n` is 123 after the group is left, unless you do this (for numerics there is actually no need to declare them):

```
begingroup ;
  save n ; numeric n ; n := 123 ;
endgroup ;
```

Given the use of MetaPost (read: MetaFont) this makes a lot of sense: often you use macros to simplify code and you do want variables to change. Grouping in this language

serves other purposes, like hiding what is between these commands and let the last expression become the result. In a `vardef` grouping is implicit.

So, in MetaPost all assignments are global, unless a variable is explicitly saved inside a group.

7.1.4 Lua

In Lua all assignments are global unless a variable is defines local:

```
local x = 1
local y = 1
for i = 1, 10 do
    local x = 2
    y = 2
end
```

Here the value of `x` after the loop is still one but `y` is now two. As in LuaTeX we mix TeX, MetaPost and Lua you can mix up these concepts. Another mixup is using `:=`, `endfor`, `fi` in Lua after done some MetaPost coding or using `end` instead of `endfor` in MetaPost which can make the library wait for more without triggering an error. Proper syntax highlighting in an editor clearly helps.

7.1.5 C

The Lua language is a mix between Pascal (which is one reason why I like it) and C.

```
int x = 1 ;
int y = 1 ;
for (i=1; i<=10;i++) {
    int x = 2 ;
    y = 2 ;
}
```

The semicolon is also used in Pascal but there it is a separator and not a statement end, while in MetaPost it does end a statement (expression).

7.2 Kinds of grouping

Explicit grouping is accomplished by the two grouping primitives:

```
\begingroup
  \sl render slanted here
\endgroup
```

However, often you will find this being used:

```
{\sl render slanted here}
```

This is not only more compact but also avoids the `\endgroup` gobbling following spaces when used inline. The next code is equivalent but also suffers from the gobbling:

```
\bgroup
  \sl render slanted here
\egroup
```

The `\bgroup` and `\egroup` commands are not primitives but aliases (made by `\let`) to the left and right curly brace. These two characters have so called category codes that signal that they can be used for grouping. The *can be* here suggest that there are other purposes and indeed there are, for instance in:

```
\toks 0 = {abs}
\hbox {def}
```

In the case of a token list assignment the curly braces fence the assignment, so scanning stops when a matching right brace is found. The following are all valid:

```
\toks 0 = {a{b}s}
\toks 0 = \bgroup a{b}s}
\toks 0 = {a{\bgroup b}s}
\toks 0 = {a{\egroup b}s}
\toks 0 = \bgroup a{\bgroup b}s}
\toks 0 = \bgroup a{\egroup b}s}
```

They have in common that the final fence has to be a right brace. That the first one can be an alias is due to the fact that the scanner searches for a brace equivalent when it looks for the value. Because the equal is optional, there is some lookahead involved which involves expansion and possibly push back while once scanning for the content starts just tokens are collected, with a fast check for nested and final braces.

In the case of the box, all these specifications are valid:

```
\hbox {def}
\hbox \bgroup def\egroup
```

Kinds of grouping

```
\hbox \bgroup def}
\hbox \bgroup d{e\egroup f}
\hbox {def\egroup
```

This is because now the braces and equivalent act as grouping symbols so as long as they match we're fine. There is a pitfall here: you cannot mix and match different grouping, so the next issues an error:

```
\bgroup xxx\endgroup % error
\begingroup xxx\egroup % error
```

This can make it somewhat hard to write generic grouping macros without trickery that is not always obvious to the user. Fortunately it can be hidden in macros like the helper `\groupedcommand`. In LuaMetaTeX we have a clean way out of this dilemma:

```
\beginsimplegroup xxx\endsimplegroup
\beginsimplegroup xxx\endgroup
\beginsimplegroup xxx\egroup
```

When you start a group with `\beginsimplegroup` you can end it in the three ways shows above. This means that the user (or calling macro) doesn't take into account what kind of grouping was used to start with.

When we are in math mode things are different. First of all, grouping with `\begin-group` and `\endgroup` in some cases works as expected, but because the math input is converted in a list that gets processed later some settings can become persistent, like changes in style or family. You can bet better use `\beginmathgroup` and `\endmathgroup` as they restore some properties. We also just mention the `\frozen` prefix that can be used to freeze assignments to some math specific parameters inside a group.

7.3 Hooks

In addition to the original `\aftergroup` primitive we have some more hooks. They can best be demonstrated with an example:

```
\begingroup \bf
%
\aftergroup A \aftergroup 1
\atendofgroup B \atendofgroup 1
%
\aftergrouped {A2}
\atendofgrouped {B2}
```

```
%
test
\endgroup
```

These collectors are accumulative. Watch how the bold is applied to what we inject before the group ends.

test B1B2A1A2

7.4 Local versus global

When \TeX enters a group and an assignment is made the current value is stored on the save stack, and at the end of the group the original value is restored. In \LuaMetaTeX this mechanism is made a bit more efficient by avoiding redundant stack entries. This is also why the next feature can give unexpected results when not used wisely.

Now consider the following example:

```
\newdimension\MyDimension

\starttabulate[||||]
  \NC      \MyDimension10pt \the\MyDimension
  \NC \advance\MyDimension10pt \the\MyDimension
  \NC \advance\MyDimension10pt \the\MyDimension \NC \NR
  \NC      \MyDimension10pt \the\MyDimension
  \NC \advance\MyDimension10pt \the\MyDimension
  \NC \advance\MyDimension10pt \the\MyDimension \NC \NR
\stoptabulate

10.0pt 10.0pt 10.0pt
10.0pt 10.0pt 10.0pt
```

The reason why we get the same values is that cells are a group and therefore the value gets restored as we move on. We can use the `\global` prefix to get around this

```
\starttabulate[||||]
  \NC \global      \MyDimension10pt \the\MyDimension
  \NC \global\advance\MyDimension10pt \the\MyDimension
  \NC \global\advance\MyDimension10pt \the\MyDimension \NC \NR
  \NC \global      \MyDimension10pt \the\MyDimension
  \NC \global\advance\MyDimension10pt \the\MyDimension
  \NC \global\advance\MyDimension10pt \the\MyDimension \NC \NR
```

\stoptabulate

```
10.0pt 20.0pt 30.0pt
10.0pt 20.0pt 30.0pt
```

Instead of using a global assignment and increment we can also use the following

\constrained\MyDimension\zeropoint

\starttabulate[||||]

```
\NC \retained \MyDimension10pt \the\MyDimension
\NC \retained\advance\MyDimension10pt \the\MyDimension
\NC \retained\advance\MyDimension10pt \the\MyDimension \NC \NR
\NC \retained \MyDimension10pt \the\MyDimension
\NC \retained\advance\MyDimension10pt \the\MyDimension
\NC \retained\advance\MyDimension10pt \the\MyDimension \NC \NR
```

\stoptabulate

```
10.0pt 20.0pt 30.0pt
10.0pt 20.0pt 30.0pt
```

So what is the difference with the global approach? Say we have these two buffers:

\startbuffer[one]

\global\MyDimension\zeropoint

\framed {

\framed {\global\advance\MyDimension10pt \the\MyDimension}

\framed {\global\advance\MyDimension10pt \the\MyDimension}

}

\framed {

\framed {\global\advance\MyDimension10pt \the\MyDimension}

\framed {\global\advance\MyDimension10pt \the\MyDimension}

}

\stopbuffer

\startbuffer[two]

\global\MyDimension\zeropoint

\framed {

\framed {\global\advance\MyDimension10pt \the\MyDimension}

\framed {\global\advance\MyDimension10pt \the\MyDimension}

\getbuffer[one]

}

\framed {

```

\framed {\global\advance\MyDimension10pt \the\MyDimension}
\framed {\global\advance\MyDimension10pt \the\MyDimension}
\getbuffer[one]
}
\stopbuffer

```

Typesetting the second buffer gives us:

10.0pt	20.0pt	10.0pt	20.0pt	30.0pt	40.0pt
50.0pt	60.0pt	10.0pt	20.0pt	30.0pt	40.0pt

When we want to have these entities independent and not use different variables, the global settings bleeding from one into the other entity is messy. Therefore we can use this:

```

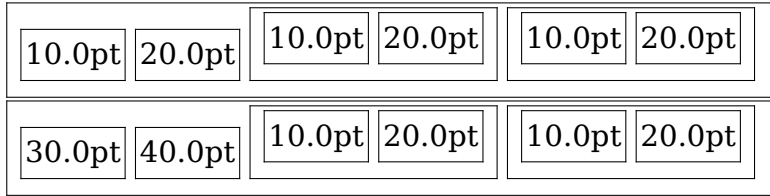
\startbuffer[one]
\constrained\MyDimension\zeropoint
\framed {
  \framed {\retained \MyDimension10pt \the\MyDimension}
  \framed {\retained\advance\MyDimension10pt \the\MyDimension}
}
\framed {
  \framed {\retained \MyDimension10pt \the\MyDimension}
  \framed {\retained\advance\MyDimension10pt \the\MyDimension}
}
\stopbuffer

\startbuffer[two]
\constrained\MyDimension\zeropoint
\framed {
  \framed {\retained\advance\MyDimension10pt \the\MyDimension}
  \framed {\retained\advance\MyDimension10pt \the\MyDimension}
  \getbuffer[one]
}
\framed {
  \framed {\retained\advance\MyDimension10pt \the\MyDimension}
  \framed {\retained\advance\MyDimension10pt \the\MyDimension}
  \getbuffer[one]
}

```

\stopbuffer

Now we get this:



The `\constrained` prefix makes sure that we have a stack entry, without being clever with respect to the current value. Then the `\retained` prefix can do its work reliably and avoid pushing the old value on the stack. Without the constrain it gets a bit unpredictable because then it all depends on where further up the chain the value was put on the stack. Of course one can argue that we should not have the “save stack redundant entries optimization” but that's not going to be removed.

7.5 Files

Although it doesn't really fit in this chapter, here are some hooks into processing files:

```
Hello World!\atendoffiled           {\writestatus{FILE}{ATEOF B #1}}\par
Hello World!\atendoffiled           {\writestatus{FILE}{ATEOF A #1}}\par
Hello World!\atendoffiled reverse {\writestatus{FILE}{ATEOF C #1}}\par
Hello World!\begingroup\sl \atendoffiled {\endgroup}\par
```

Inside a file you can register tokens that will be expanded when the file ends. You can also do that beforehand using a variant of the `\input` primitive:

```
\eofinput {\writestatus{FILE}{DONE}} {thatfile.tex}
```

This feature is mostly there for consistency with the hooks into groups and paragraphs but also because `\everyeof` is kind of useless given that one never knows beforehand if a file loads another file. The hooks mentioned above are bound to the current file.

7.5 Colofon

Author	Hans Hagen
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com
	contextgarden.net
	ntg-context@ntg.nl

8 Security

The background of the entire image is a solid brown color with a repeating pattern of thin, dark gray diagonal lines running from the top-left to the bottom-right.

low level

TEX

security

Contents

8.1	Preamble	143
8.2	Flags	143
8.3	Complications	146
8.4	Introspection	147

8.1 Preamble

Here I will discuss a moderate security subsystem of LuaMetaT_EX and therefore ConT_EXt LMTX. This is not about security in the sense of the typesetting machinery doing harm to your environment, but more about making sure that a user doesn't change the behavior of the macro package in ways that introduce interference and thereby unwanted side effect. It's all about protecting macros.

This is all very experimental and we need to adapt the ConT_EXt source code to this. Actually that will happen a few times because experiments trigger that. It might take a few years before the security model is finalized and all files are updated accordingly. There are lots of files and macros involved. In the process the underlying features in the engine might evolve.

8.2 Flags

Before we go into the security levels we see what flags can be set. The T_EX language has a couple of so called prefixes that can be used when setting values and defining macros. Any engine that has traditional T_EX with ε -T_EX extensions can do this:

```

\def\foo{foo}
\global\def\foo{foo}
\global\protected\def\foo{foo}

```

And LuaMetaT_EX adds another one:

```

\tolerant\def\foo{foo}
\global\tolerant\def\foo{foo}
\global\tolerant\protected\def\foo{foo}

```

What these prefixes do is discussed elsewhere. For now it is enough to know that the two optional prefixes `\protected` and `\tolerant` make for four distinctive cases of macro calls.

But there are more prefixes:

frozen	a macro that has to be redefined in a managed way
permanent	a macro that had better not be redefined
primitive	a primitive that normally will not be adapted
immutable	a macro or quantity that cannot be changed, it is a constant
mutable	a macro that can be changed no matter how well protected it is
instance	a macro marked as (for instance) be generated by an interface
noaligned	the macro becomes acceptable as \noalign alias
overloaded	when permitted the flags will be adapted
enforced	all is permitted (but only in zero mode or ini mode)
aliased	the macro gets the same flags as the original

These prefixed set flags to the command at hand which can be a macro but basically any control sequence.

To what extent the engine will complain when a property is changed in a way that violates the above depends on the parameter \overloadmode. When this parameter is set to zero no checking takes place. More interesting are values larger than zero. If that is the case, when a control sequence is flagged as mutable, it is always permitted to change. When it is set to immutable one can never change it. The other flags determine the kind of checking done. Currently the following overload values are used:

		immutable	permanent	primitive	frozen	instance
1	warning	*	*	*		
2	error	*	*	*		
3	warning	*	*	*	*	
4	error	*	*	*	*	
5	warning	*	*	*	*	*
6	error	*	*	*	*	*

The even values (except zero) will abort the run. In ConT_EXt we plug in a callback that deals with the messages. A value of 255 will freeze this parameter. At level five and above the instance flag is also checked but no drastic action takes place. We use this to signal to the user that a specific instance is redefined (of course the definition macros can check for that too).

So, how does it work. The following is okay:

```
\def\MacroA{A}
\def\MacroB{B}
```

```
\let\MyMacro\MacroA
\let\MyMacro\MacroB
```

The first two macros are ordinary ones, and the last two lines just create an alias. Such an alias shares the definition, but when for instance `\MacroA` is redefined, its new meaning will not be reflected in the alias.

```
\permanent\protected\def\MacroA{A}
\permanent\protected\def\MacroB{B}
\let\MyMacro\MacroA
\let\MyMacro\MacroB
```

This also works, because the `\let` will create an alias with the protected property but it will not take the permanent property along. For that we need to say:

```
\permanent\protected\def\MacroA{A}
\permanent\protected\def\MacroB{B}
\permanent\let\MyMacro\MacroA
\permanent\let\MyMacro\MacroB
```

or, when we want to copy all properties:

```
\permanent\protected\def\MacroA{A}
\permanent\protected\def\MacroB{B}
\aliased\let\MyMacro\MacroA
\aliased\let\MyMacro\MacroB
```

However, in `ConTEXt` we have commands that we like to protect against overloading but at the same time have a different meaning depending on the use case. An example is the `\NC` (next column) command that has a different implementation in each of the table mechanisms.

```
\permanent\protected\def\NC_in_table {...}
\permanent\protected\def\NC_in_tabulate{...}
\aliased\let\NC\NC_in_table
\aliased\let\NC\NC_in_tabulate
```

Here the second aliasing of `\NC` fails (assuming of course that we enabled overload checking). One can argue that grouping can be used but often no grouping takes place when we redefine on the fly. Because `frozen` is less restrictive than `primitive` or `permanent`, and of course `immutable`, the next variant works:

```
\frozen\protected\def\NC_in_table {...}
```

```
\frozen\protected\def\NC_in_tabulate{...}
\overloaded\let\NC\NC_in_table
\overloaded\let\NC\NC_in_tabulate
```

However, in practice, as we want to keep the overload checking, we have to do:

```
\frozen\protected\def\NC_in_table {...}
\frozen\protected\def\NC_in_tabulate{...}
\overloaded\frozen\let\NC\NC_in_table
\overloaded\frozen\let\NC\NC_in_tabulate
```

or use `\aliased`, but there might be conflicting permissions. This is not that nice, so there is a kind of dirty trick possible. Consider this:

```
\frozen\protected\def\NC_in_table {...}
\frozen\protected\def\NC_in_tabulate{...}
\def\setNCintable {\enforced\let\frozen\let\NC\NC_in_table}
\def\setNCintabulate{\enforced\let\frozen\let\NC\NC_in_tabulate}
```

When we're in so called `initex` mode or when the overload mode is zero, the `\enforced` prefix is internalized in a way that signals that the follow up is not limited by the overload mode and permissions. This definition time binding mechanism makes it possible to use permanent macros that users cannot redefine, but existing macros can, unless of course they tweak the mode parameter.

Now keep in mind that users can always cheat but that is intentional. If you really want to avoid that you can set the overload mode to 255 after which it cannot be set any more. However, it can be useful to set the mode to zero (or some warning level) when foreign macro packages are used.

8.3 Complications

One side effect of all this is that all those prefixes can lead to more code. On the other hand we save some due to the extended macro argument handling features. When you take the size of the format file as reference, in the end we get a somewhat smaller file. Every token that you add or remove gives a 8 bytes difference. The extra overhead that got added to the engine is compensated by the fact that some macro implementations can be more efficient. In the end, in spite of these new features and the more extensive testing of flags performance is about the same.¹⁰

¹⁰ And if you wonder about memory, by compacting the used (often scattered) token memory before dumping I manages to save some 512K on the format file, so often the loss and gain are somewhere else.

8.4 Introspection

In case you want to get some details about the properties of a macro, you can check its meaning. The full variant shows all of them.

% a macro with two optional arguments with optional spacing in between:

```
\permanent\tolerant\protected\def\MyFoo[#1]#*[#2]{(#1)(#2)}
```

```
\meaningless\MyFoo\par
```

```
\meaning      \MyFoo\par
```

```
\meaningfull\MyFoo\par
```

```
[#1]#*[#2]->(#1)(#2)
```

```
tolerant protected macro:[#1]#*[#2]->(#1)(#2)
```

```
permanent tolerant protected macro:[#1]#*[#2]->(#1)(#2)
```

8.4 Colofon

Author	Hans Hagen
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com
	contextgarden.net
	ntg-context@ntg.nl

9 Characters



low level

TEX

characters

Contents

9.1	Introduction	149
9.2	History	149
9.3	The heritage	150
9.4	The LMTX approach	151
9.5	spaces	154

9.1 Introduction

This explanation is part of the low level manuals because in practice users will not have to deal with these matters in MkIV and even less in LMTX. You can skip to the last section for commands.

9.2 History

If we travel back in time to when T_EX was written we end up in eight bit character universe. In fact, the first versions assumed seven bits, but for comfortable use with languages other than English that was not sufficient. Support for eight bits permits the usage of so called code pages as supported by operating systems. Although ascii input became kind of the standard soon afterwards, the engine can be set up for different encodings. This is not only true for T_EX, but for many of its companions, like MetaFont and therefore MetaPost.¹¹

Core components of a T_EX engine are hyphenation of words, applying inter-character kerns and build ligatures. In traditional T_EX engines those processes are interwoven into the par builder but in LuaT_EX these are separate stages. The original approach is the reason that there is a relation between the input encoding and the font encoding: the character in the input is the slot used in a reference to a glyph. When producing the final result (e.g. pdf) there can also be a mapping to an index in a font resource.

input A [tex ->] font slot A [backend ->] glyph index A

The mapping that T_EX does is normally one-to-one but an input character can undergo some transformation. For instance a character beyond ascii 126 can be made active and expand to some character number that then becomes the font slot. So, it is the

¹¹ This remapping to an internal representation (e.g. ebcdic) is not present in LuaT_EX where we assume utf8 to be the input encoding. The MetaPost library that comes with LuaT_EX still has that code but in LuaMetaT_EX it's gone. There one can set up the machinery to be utf8 aware too.

expansion (or meaning) of a character that end up as numeric reference in the glyph node. Virtual fonts can introduce yet another remapping but that's only visible in the backend.

Actually, in LuaT_EX the same happens but in practice there is no need to go active because (at least in ConT_EXt) we assume a Unicode path so there the font slot is the Unicode got from the utf8 input.

In the eight bit universe macro packages (have to) provide all kind of means to deal with (in the perspective of English) special characters. For instance, `\"a` would put a diaeresis on top of the a or even better, refer to a character in the encoding that the chosen font provides. Because there are some limitations of what can go in an eight bit font, and because in different countries the used T_EX fonts evolved kind of independent, we ended up with quite some different variants of fonts. It was only with the Latin Modern project that this became better. Interesting is that when we consider the fact that such a font has often also hardly used symbols (like registered or copyright) coming up with an encoding vector that covers most (latin based) European languages (scripts) is not impossible¹² Special symbols could simply go into a dedicated font, also because these are always accessed via a macro so who cares about the input. It never happened.

Keep in mind that when utf8 is used with eight bit engines, ConT_EXt will convert sequences of characters into a slot in a font (depending on the font encoding used which itself depends on the coverage needed). For this every first (possible) byte of a multi-byte utf sequence is an active character, which is no big deal because these are outside the ascii range. Normal ascii characters are single byte utf sequences and fall through without treatment.

Anyway, in ConT_EXt MkII we dealt with this by supporting mixed encodings, depending on the (local) language, referencing the relevant font. It permits users to enter the text in their preferred input encoding and also get the words properly hyphenated. But we can leave these MkII details behind.

9.3 The heritage

In MkIV we got rid of input and font encodings, although one can still load files in a specific code page.¹³ We also kept the means to enter special characters, if only because

¹² And indeed in the Latin Modern project we came up with one but it was already to late for it to become popular.

¹³ I'm not sure if users ever depend on an input encoding different from utf8.

text editors seldom support(ed) a wide range of visual editing of those. This is why we still have

```
"u ^a v{s} \AE \ij \eacute \oslash
```

and many more. The ones with one character names are rather common in the T_EX community but it is definitely a weird mix of symbols. The next two are kind of outdated: in these days you delegate that to the font handler, where turning them into ‘single’ character references depends on what the font offers, how it is set up with respect to (for instance) ligatures, and even might depend on language or script.

The ones with the long names partly are tradition, but as we have a lot of them, in MkII they actually serve a purpose. These verbose names are used in the so called encoding vectors and are part of the utf expansion vectors. They are also used in labels so that we have a good indication if what goes in there: remember that in those times editors often didn't show characters, unless the font for display had them, or the operating system somehow provided them from another font. These verbose names are used for latin, greek and cyrillic and for some other scripts and symbols. They take up quite a bit of hash space and the format file.¹⁴

9.4 The LMTX approach

In the process of tagging all (public) macros in LMTX (which happened in 2020-2021) I wondered if we should keep these one character macros, the references to special characters and the verbose ones. When asked on the mailing list it became clear that users still expect the short ones to be present, often just because old bibT_EX files are used that might need them. However, in MkIV and LMTX we load bibT_EX files in a way that turn these special character references into proper utf8 input so it makes a weak argument. Anyway, although they could go, for now we keep them because users expect them. However, in LMTX the implementation is somewhat different now, a bit more efficient in terms of hash and memory, potentially a bit less efficient in runtime, but no one will notice that.

A new command has been introduced, the very short `\chr`.

```
\chr {a} \chr {a} \chr {a}
\chr {'a} \chr {'a} \chr {"a}
\chr {a acute} \chr {a grave} \chr {a umlaut}
```

¹⁴ In MkII we have an abstract front-end with respect to encodings and also an abstract backend with respect to supported drivers but both approaches no longer make sense today.

```
\chr {aacute} \chr {agrave} \chr {aumlaut}
```

In the first line the composed character using two characters, a base and a so called mark. Actually, one doesn't have to use `\chr` in that case because `ConTEXt` does already collapse characters for you. The second line looks like the shortcuts `\``, `\'` and `\"`. The third and fourth lines could eventually replace the more symbolic long names, if we feel the need. Watch out: in Unicode input the marks come *after*.

```
à á â
à á â
á à a~mla~t
á à a~mla~t
```

Currently the repertoire is somewhat limited but it can be easily be extended. It all depends on user needs (doing Greek and Cyrillic for instance). The reason why we actually save code deep down is that the helpers for this have always been there.¹⁵

The `\"` commands are now just aliases to more verbose and less hackery looking macros:

<code>\withgrave</code>	à	<code>\`</code>	à
<code>\withacute</code>	á	<code>\'</code>	á
<code>\withcircumflex</code>	â	<code>\^</code>	â
<code>\withtilde</code>	ã	<code>\~</code>	ã
<code>\withmacron</code>	ā	<code>\=</code>	ā
<code>\withbreve</code>	ĕ	<code>\u</code>	ĕ
<code>\withdotaccent</code>	ċ	<code>\.</code>	.c
<code>\withdiaeresis</code>	ë	<code>\"</code>	ë
<code>\withring</code>	ŭ	<code>\r</code>	ŭ
<code>\withhungarumlaut</code>	ű	<code>\H</code>	ű
<code>\withcaron</code>	ě	<code>\v</code>	ě
<code>\withcedilla</code>	ç	<code>\c</code>	ç
<code>\withogonek</code>	ę	<code>\k</code>	ę

Not all fonts have these special characters. Most natural is to have them available as precomposed single glyphs, but it can be that they are just two shapes with the marks anchored to the base. It can even be that the font somehow overlays them, assuming (roughly) equal widths. The compose font feature in `ConTEXt` normally can handle most well.

¹⁵ So if needed I can port this approach back to MkIV, but for now we keep it as is because we then have a reference.

An occasional ugly rendering doesn't matter that much: better have something than nothing. But when it's the main language (script) that needs them you'd better look for a font that handles them. When in doubt, in ConT_EXt you can enable checking:

command	equivalent to
<code>\checkmissingcharacters</code>	<code>\enabletrackers[fonts.missing]</code>
<code>\removemissingcharacters</code>	<code>\enabletrackers[fonts.missing=remove]</code>
<code>\replacemissingcharacters</code>	<code>\enabletrackers[fonts.missing=replace]</code>
<code>\handlemissingcharacters</code>	<code>\enabletrackers[fonts.missing={decompose,replace}]</code>

The decompose variant will try to turn a composed character into its components so that at least you get something. If that fails it will inject a replacement symbol that stands out so that you can check it. The console also mentions missing glyphs. You don't need to enable this by default¹⁶ but you might occasionally do it when you use a font for the first time.

In LMTX this mechanism has been upgraded so that replacements follow the shape and are actually real characters. The decomposition has not yet been ported back to MkIV.

The full list of commands can be queried when a tracing module is loaded:

```
\usemodule[s][characters-combinations]
```

```
\showcharactercombinations
```

We get this list:

acute	U+00301	´	<code>\withacute</code>
breve	U+00306	˘	<code>\withbreve</code>
caron	U+0030C	ˇ	<code>\withcaron</code>
caron below	U+0032C	ˇ̣	<code>\withcaronbelow</code>
cedilla	U+00327	¸	<code>\withcedilla</code>
circumflex	U+00302	ˆ	<code>\withcircumflex</code>
circumflex below	U+0032D	ˆ̣	<code>\withcircumflexbelow</code>
comma below	U+00326	¸̣	<code>\withcommabelow</code>
diaeresis	U+00308	¨	<code>\withdiaeresis</code>
dieresis	U+00308	¨	<code>\withdieresis</code>
dot	U+00307	˙	<code>\withdot</code>
dot below	U+00323	˙̣	<code>\withdotbelow</code>
double acute	U+0030B	˝	<code>\withdoubleacute</code>

¹⁶ There is some overhead involved here.

double grave	U+0030F	“	\withdoublegrave
double vertical line	U+0030E	”	\withdoubleverticalline
grave	U+00300	`	\withgrave
hook	U+00309	’	\withhook
hook below	U+1FA9D		\withhookbelow
hungarumlaut	U+0030B	“	\withhungarumlaut
inverted breve	U+00311	˘	\withinvertedbreve
line	U+00304	-	\withline
line below	U+00331	_	\withlinebelow
macron	U+00304	-	\withmacron
macron below	U+00331	_	\withmacronbelow
middle dot	U+000B7	·	\withmiddledot
ogonek	U+00328	˙	\withogonek
overline	U+00305	—	
ring	U+0030A	°	\withring
ring below	U+00325	◌	\withringbelow
slash	U+0002F	/	\withslash
stroke	U+0002F	/	\withstroke
tilde	U+00303	~	\withtilde
tilde below	U+00330	˜	\withtildebelow
vertical line	U+0030D	’	\withverticalline

Some combinations are special for ConT_EXt because Unicode doesn't specify decomposition for all composed characters.

9.5 spaces

The engine has no real concept of a space. When the input has one it is turned into a glue, likely with some stretch and shrink. When `\nospaces` is set to one, no glue will be inserted. A value of two will inject a zero width glue. When set to three a glyph will be inserted with the character code set by `\spacechar`.

`\nospaces\plusthree`

`\spacechar\underscoreasciicode`

`\hccode\underscoreasciicode\underscoreasciicode`

Where are the spaces?

The `hccode` tells the machinery that the underscore is a valid word separator (think compound words).

Where_are_the_spaces?

9.5 Colofon

Author	Hans Hagen
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com contextgarden.net ntg-context@ntg.nl

10 Scope

low level

TEX

scope

Contents

10.1	Introduction	157
10.2	Registers	157
10.3	Allocation	159
10.4	Files	162

10.1 Introduction

When I visited the file where register allocations are implemented I wondered to what extend it made sense to limit allocation to global instances only. This chapter deals with this phenomena.

10.2 Registers

In \TeX definitions can be local or global. Most assignments are local within a group. Register

```
\scratchcounter = 1
here the counter has value 1
\begingroup
  \scratchcounter = 2
  here the counter has value 2
\endgroup
here the counter has value 1
```

with:

```
\setbox\scratchbox=\hbox{}
here the box has zero width
\begingroup
  \wd\scratchbox=10pt
  here the box is 10pt wide
\endgroup
here the box is 10pt wide
```

It all makes sense so a remark like “Assignments to box dimensions are always global” are sort of confusing. Just look at this:

```
\setbox\scratchbox=\hbox to 20pt{}
here the box is \the\wd\scratchbox\ wide\par
```

```

\begingroup
  \setbox\scratchbox=\hbox{}
  here the box is \the\wd\scratchbox\ wide\par
  \begingroup
    \wd\scratchbox=15pt
    here the box is \the\wd\scratchbox\ wide\par
  \endgroup
  here the box is \the\wd\scratchbox\ wide\par
\endgroup
here the box is \the\wd\scratchbox\ wide\par

```

```

here_the_box_is_20.0pt_wide
here_the_box_is_0.0pt_wide
here_the_box_is_15.0pt_wide
here_the_box_is_15.0pt_wide
here_the_box_is_20.0pt_wide

```

If you don't think about it, what happens is what you expect. Now watch the next variant:

The `\global` is only effective for the current box. It is good to realize that when we talk registers, the box register behaves just like any other register but the manipulations happen to the current one.

```

\setbox\scratchbox=\hbox to 20pt{}
here the box is \the\wd\scratchbox\ wide\par
\begingroup
  \setbox\scratchbox=\hbox{}
  here the box is \the\wd\scratchbox\ wide\par
  \begingroup
    \global\wd\scratchbox=15pt
    here the box is \the\wd\scratchbox\ wide\par
  \endgroup
  here the box is \the\wd\scratchbox\ wide\par
\endgroup
here the box is \the\wd\scratchbox\ wide\par

```

```

here_the_box_is_20.0pt_wide
here_the_box_is_0.0pt_wide
here_the_box_is_15.0pt_wide
here_the_box_is_15.0pt_wide
here_the_box_is_20.0pt_wide

```

```

\scratchdimen=20pt
here the dimension is \the\scratchdimen\par
\begingroup
  \scratchdimen=0pt
  here the dimension is \the\scratchdimen\par
  \begingroup
    \global\scratchdimen=15pt
    here the dimension is \the\scratchdimen\par
  \endgroup
  here the dimension is \the\scratchdimen\par
\endgroup
here the dimension is \the\scratchdimen\par

here_the_dimension_is_20.0pt
here_the_dimension_is_0.0pt
here_the_dimension_is_15.0pt
here_the_dimension_is_15.0pt
here_the_dimension_is_15.0pt

```

10.3 Allocation

The plain TeX format has set some standards and one of them is that registers are allocated

```

\newcount\mycounta
\newdimen\mydimena

```

These commands take a register from the pool and relate the given name to that entry. In ConTeXt we have a bunch of predefined scratch registers for general use, like:

```

scratchcounter      : \meaningfull\scratchcounter
scratchcounterone   : \meaningfull\scratchcounterone
scratchcountertwo    : \meaningfull\scratchcountertwo
scratchdimen        : \meaningfull\scratchdimen
scratchdimenone      : \meaningfull\scratchdimenone
scratchdimentwo      : \meaningfull\scratchdimentwo

```

The meaning reveals what these are:

```

scratchcounter : _global_constant_integer_1026
scratchcounterone : _global_constant_integer_0
scratchcountertwo : _global_constant_integer_0
scratchdimen : _global_constant_dimension_15.0pt

```

```
scratchdimenone:_global_constant_dimension_0.0pt
scratchdimentwo:_global_constant_dimension_0.0pt
```

You can use the numbers directly but that is a bad idea because they can clash! In the original \TeX engine there are only 256 registers and some are used by the engine and \TeX upped that to 64K. One could go higher but what makes sense? These registers are taken

As mentioned, commands like `\newcount\foo` create a global control sequence `\foo` referencing a counter. You can locally redefine that control sequence unless in LuaMeta- \TeX you have so called overload mode enabled. You can do local or global assignments to th

```
\scratchcounter = 123
\begingroup
  \scratchcounter = 456
  \begingroup
    \global\scratchcounter = 789
  \endgroup
\endgroup
```

And in both cases count register 257 is set. When an assignment is global, all current values to that register get the same value. Normally this is all quite transparent: you get what you ask for. However the drawback is that as a user you cannot know what variables are already defined, which means that this will fail (that is: it will issue a message):

```
\newcount\scratchcounter
```

as will the second line in:

```
\newcount\myscratchcounter
\newcount\myscratchcounter
```

In \ConTeXt the scratch registers are visible but there are lots of internally used ones are \pr \TeX barking to you about not being able to define it. This is why in LMTX (and maybe some

```
\begingroup
  \newlocaldimen\mydimena    \mydimena1\onepoint
  \newlocaldimen\mydimenb    \mydimenb2\onepoint
  (\the\mydimena,\the\mydimenb)
  \begingroup
    \newlocaldimen\mydimena    \mydimena3\onepoint
    \newlocaldimen\mydimenb    \mydimenb4\onepoint
    \newlocaldimen\mydimenc    \mydimenc5\onepoint
```

```

(\the\mydimena,\the\mydimenb,\the\mydimenc)
\begingroup
  \newlocaldimen\mydimena \mydimena6\onepoint
  \newlocaldimen\mydimenb \mydimenb7\onepoint
  (\the\mydimena,\the\mydimenb)
\endgroup
\newlocaldimen\mydimend \mydimend8\onepoint
(\the\mydimena,\the\mydimenb,\the\mydimenc,\the\mydimend)
\endgroup
(\the\mydimena,\the\mydimenb)
\endgroup

```

The allocated registers get zero values but you can of course set them to any value that fits their nature:

```

(1.0pt,2.0pt)
(3.0pt,4.0pt,5.0pt)
(6.0pt,7.0pt)
(3.0pt,4.0pt,5.0pt,8.0pt)
(1.0pt,2.0pt)

```

You can also use the next variant where you also pass the initial value:

```

\begingroup
  \setnewlocaldimen\mydimena 1\onepoint
  \setnewlocaldimen\mydimenb 2\onepoint
  (\the\mydimena,\the\mydimenb)
\begingroup
  \setnewlocaldimen\mydimena 3\onepoint
  \setnewlocaldimen\mydimenb 4\onepoint
  \setnewlocaldimen\mydimenc 5\onepoint
  (\the\mydimena,\the\mydimenb,\the\mydimenc)
\begingroup
  \setnewlocaldimen\mydimena 6\onepoint
  \setnewlocaldimen\mydimenb 7\onepoint
  (\the\mydimena,\the\mydimenb)
\endgroup
\setnewlocaldimen\mydimend 8\onepoint
(\the\mydimena,\the\mydimenb,\the\mydimenc,\the\mydimend)
\endgroup
(\the\mydimena,\the\mydimenb)

```

\endgroup

So, again we get:

```
(1.0pt,2.0pt)
(3.0pt,4.0pt,5.0pt)
(6.0pt,7.0pt)
(3.0pt,4.0pt,5.0pt,8.0pt)
(1.0pt,2.0pt)
```

When used in the body of the macro there is of course a little overhead involved in the repetitive allocation but normally that can be neglected.

10.4 Files

When adding these new allocators I also wondered about the read and write allocators. We don't use them in ConT_EXt but maybe users like them, so let's give an example and

\integerdef\StartHere\numexpr\inputlineno+2\relax

\starthiding

```
SOME LINE 1
SOME LINE 2
SOME LINE 3
SOME LINE 4
```

\stophiding

\integerdef\StopHere\numexpr\inputlineno-2\relax

\begingroup

\newlocalread\myreada

\immediate\openin\myreada {lowlevel-scope.tex}

\dostepwiserecurse{\StopHere}{\StartHere}{-1}{

\readline\myreada line #1 to \scratchstring #1 : \scratchstring \par

}

\blank

\dostepwiserecurse{\StartHere}{\StopHere}{1}{

\read \myreada line #1 to \scratchstring #1 : \scratchstring \par

}

\immediate\closein\myreada

\endgroup

Here, instead of hard coded line numbers we used the stored values. The optional line keyword is a LMTX speciality.

```

281_: SOME_LINE_4
280_: SOME_LINE_3
279_: SOME_LINE_2
278_: SOME_LINE_1

278_: SOME_LINE_1_
279_: SOME_LINE_2_
280_: SOME_LINE_3_
281_: SOME_LINE_4_

```

Actually an application can be found in a small (demonstration) module:

```
\usemodule[system-readers]
```

This provides the code for doing this:

```

\startmarkedlines[test]
SOME LINE 1
SOME LINE 2
SOME LINE 3
\stopmarkedlines

```

```
\begingroup
```

```
  \newlocalread\myreada
```

```
  \immediate\openin\myreada {\markedfilename{test}}
```

```
  \dostepwiserecurse{\lastmarkedline{test}}{\firstmarkedline{test}}{-1}{
    \readline\myreada line #1 to \scratchstring #1 : \scratchstring \par
  }
```

```
  \immediate\closein\myreada
```

```
\endgroup
```

As you see in these examples, we can locally define a read channel without getting a message about it already being defined.

10.4 Colofon

Author	Hans Hagen
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com
	contextgarden.net
	ntg-context@ntg.nl

11 Paragraphs



low level

TEX

paragraphs

Contents

11.1	Introduction	165
11.2	Paragraphs	165
11.3	Properties	168
11.4	Wrapping up	171
11.5	Hanging	171
11.6	Shapes	171
11.7	Modes	190
11.8	Leaders	191
11.9	Prevdepth	197
11.10	Normalization	199
11.11	Dirty tricks	199
11.12	Penalties	199
11.13	Par passes	200

11.1 Introduction

This manual is mostly discussing a few low level wrappers around low level \TeX features. It MetaFun manuals where we mess a bit with shapes. It gave a good reason to also cover so

Because paragraphs and their construction are rather central to \TeX , you can imagine that \TeX t. Intercepting and tweaking paragraph properties is even more tricky, which is why we

11.2 Paragraphs

Before we demonstrate some trickery, let's see what a paragraph is. Normally a document source is formatted like this:

```
some text (line 1)
some text (line 2)

some more test (line 1)
some more test (line 2)
```

¹⁷ For this we have `\enableexperiments` which one can use in `cont-loc.mkxl` or `cont-exp.mkxl`, files that are loaded runtime when on the system. When you use them, make sure they don't interfere; they are not part of the updates, contrary to `cont-new.mkxl`.

There are two blocks of text here separated by an empty line and they become two paragraphs. Unless configured otherwise an empty line is an indication that we end a paragraph. You can also explicitly do that:

```
some text (line 1)
some text (line 2)
\par
some more test (line 1)
some more test (line 2)
```

When \TeX starts a paragraph, it actually also does something think of:

```
[\the\everypar]some text      (line 1) some text      (line 2) \par
[\the\everypar]some more test (line 1) some more test (line 2) \par
```

or more accurate:

```
[\the\everypar]some text      some text      \par
[\the\everypar]some more test some more test \par
```

because the end-of-line character has become a space. As mentioned, an empty line is actually the end of a paragraph. But in LuaMeta \TeX we can cheat a bit. If we have this:

line 1

line 2

We can do this (watch how we need to permit overloading a primitive when we have enabled `\overloadmode`):

```
\pushoverloadmode
\def\linepar{\removeunwantedspaces !\ignorespaces}
\popoverloadmode
line 1

line 2
```

This comes out as:

line_1_

line_2_

I admit that since it got added (as part of some cleanup halfway the overhaul of the engine) I never saw a reason to use it, but it is a cheap feature. The `\linepar` primitive is undefined (`\undefined`) by default so no user sees it anyway. Just don't use it unless maybe for some pseudo database trickery (I considered using it for the database module but it is not needed). In a similar fashion, just don't redefine `\par`: it's asking for troubles and 'not done' in ConTeXt anyway.

Back to reality. In LuaTeX we get a node list that starts with a so called `localpar` node and

When the paragraph is broken into lines hanging indentation or a so called `par` shape can be applied, and we will see more of that later, here we talk `\par` and show another LuaMetaTeX trick:

```
\def\foo{\bf test:} \ignorepars}
```

```
\foo
```

```
line
```

The macro typesets some text and then skips to the next paragraph:

```
test: line
```

Think of this primitive as being a more powerful variant of `\ignorespaces`. This leaves one aspect: how do we start a paragraph. Technically we need to force TeX into so called horizontal mode. In TeX we have more high level variants, for instance we have `\noindentation`.

A robust way to make sure that you get in horizontal mode is using `\dontleavehmode` which is a wink to `\leavevmode`, a command that you should never use in ConTeXt, so when you c

When TeX starts with a paragraph the `\everypar` token list is expanded and again this is a

One of the things that TeX does in injecting the indentation. Even when there is none, it ge
TeXt we enable the LuaMetaTeX feature that does use a skip instead of a box. It's part of th

A bit off topic is the fact that in traditional TeX empty lines or `\par` primitives can trigger a TeXt MkII most macros that could be sensitive for this were defined as `\long` so that users TeX these error-triggers could be disabled which of course we enable in ConTeXt and in Lua MetaTeX these features have been removed altogether. I don't think users will complain abo

If you want to enforce a newline but not a new paragraph you can use the `\crlf` command. When used on its own it will produce an empty line. Don't use this to create whitespace between lines.

If you want to do something after so-called par tokens are seen you can do this:

```
\def\foo{{\bf >>>> }}
\expandafterpars\foo
```

this is a new paragraph ...

```
\expandafterpars\foo
\par\par\par\par
```

this is a new paragraph ...

This not to be confused with `\everypar` which is a token list that `TEX` itself injects before e

```
>>>>_this_is_a_new_paragraph_..._
```

```
>>>>_this_is_a_new_paragraph_..._
```

This is typically a primitive that will only be used in macros. You can actually program it using macros: pickup a token, check and push it back when it's not a par equivalent token. The primitive is just nicer (and easier on the log when tracing is enabled).

11.3 Properties

A paragraph is just a collection of lines that result from one input line that got broken. This process of breaking into lines is influenced by quite some parameters. In traditional `TEX` and also in `LuaMetaTEX` by default the values that are in effect when the en

However, in `LuaMetaTEX` we can optionally store them with the paragraph. When that happo

variable	category	code
<code>\hsize</code>	hsize	0x00000001
<code>\leftskip</code>	skip	0x00000002
<code>\rightskip</code>	skip	0x00000002
<code>\hangindent</code>	hang	0x00000004
<code>\hangafter</code>	hang	0x00000004
<code>\parindent</code>	indent	0x00000008
<code>\parfilllleftskip</code>	parfill	0x00000010
<code>\parfillskip</code>	parfill	0x00000010
<code>\parinitleftskip</code>	parfill	0x00000010
<code>\parinitrightskip</code>	parfill	0x00000010
<code>\emergencyleftskip</code>	emergency	0x00800000

\emergencyrightskip	emergency	0x00800000
\adjustspacing	adjust	0x00000020
\protrudechars	protrude	0x00000040
\pretolerance	tolerance	0x00000080
\tolerance	tolerance	0x00000080
\emergencystretch	stretch	0x00000100
\looseness	looseness	0x00000200
\lastlinefit	lastline	0x00000400
\linepenalty	linepenalty	0x00000800
\interlinepenalty	linepenalty	0x00000800
\clubpenalty	clubpenalty	0x00001000
\widowpenalty	widowpenalty	0x00002000
\displaywidowpenalty	displaypenalty	0x00004000
\lefttwindemerits	twindemerits	0x20000000
\righttwindemerits	twindemerits	0x20000000
\brokenpenalty	brokenpenalty	0x00008000
\adjdemerits	demerits	0x00010000
\doublehyphendemerits	demerits	0x00010000
\finalhyphendemerits	demerits	0x00010000
\parshape	shape	0x00020000
\interlinepenalties	linepenalty	0x00000800
\clubpenalties	clubpenalty	0x00001000
\widowpenalties	widowpenalty	0x00002000
\displaywidowpenalties	displaypenalty	0x00004000
\brokenpenalties	brokenpenalty	0x00008000
\orphanpenalties	orphanpenalty	0x00200000
\toddlrpenalties	toddlerpenalty	0x00400000
\fitnessclasses	fitnessclasses	0x40000000
\adjacentdemerits	demerits	0x00010000
\mathleftclass	orphanpenalty	0x00200000
\baselineskip	line	0x00040000
\lineskip	line	0x00040000
\lineskiplimit	line	0x00040000
\adjustspacingstep	adjust	0x00000020
\adjustspacingshrink	adjust	0x00000020
\adjustspacingstretch	adjust	0x00000020
\hyphenationmode	hyphenation	0x00080000
\shapingpenaltiesmode	shapingpenalty	0x00100000
\shapingpenalty	shapingpenalty	0x00100000
\emergencyextrastretch	emergency	0x00800000

Properties

<code>\parpasses</code>	<code>parpasses</code>	<code>0x01000000</code>
<code>\linebreakchecks</code>	<code>linebreakchecks</code>	<code>0x10000000</code>
<code>\singlelinepenalty</code>	<code>singlelinepenalty</code>	<code>0x02000000</code>
<code>\hyphenpenalty</code>	<code>hyphenpenalty</code>	<code>0x04000000</code>
<code>\exhyphenpenalty</code>	<code>exhyphenpenalty</code>	<code>0x08000000</code>

As you can see here, there are more paragraph related parameters than in for instance pdfTeX and LuaTeX and these are (to be) explained in the LuaMetaTeX manual. You can imagine

This is pretty low level and there are a bunch of helpers that support this but these are not really user level macros. As with everything TeX you can mess around as much as you want with TeX's core functionality.

In LMTX taking these snapshots is turned on by default and because it thereby fundamentally influences the par builder, users can run into compatibility issues but in practice there has been no complaints (and this feature has been in use quite a while before this document was written). One reason for users not noticing is that one of the big benefits is probably handled by tricks mentioned on the mailing list. Imagine that you have this:

```
{\bf watch out:} here is some text
```

In this small example the result will be as expected. But what if something magic with the start of a paragraph is done? Like this:

```
\placefigure[left]{A cow!}{\externalfigure[cow.pdf]}
```

```
{\bf watch out:} here is some text ... of course much more is needed to
get a flow around the figure!
```

The figure will hang at the left side of the paragraph but it is put there when the text starts and that happens inside the bold group. It means that the properties we set in order to get the shape around the figure are lost as soon as we're at 'here is some text' and definitely is wrong when the paragraph ends and the par builder has to use them to get the shape right. We get text overlapping the figure. A trick to overcome this is:

```
\dontleavehmode {\bf watch out:} here is some text ... of course much
more is needed to get a flow around the figure!
```

where the first macro makes sure we already start a paragraph before the group is entered (using a `\strut` also works). It's not nice and I bet users have been bitten by this and by now know the tricks. But, with snapshots such fuzzy hacks are not needed any more! The same is true with this:

```
{\leftskip 1em some text \par}
```

where we had to explicitly end the paragraph inside the group in order to retain the skip. I suppose that users normally use the high level environments so they never had to worry about this. It's also why users probably won't notice that this new mechanism has been active for a while. Actually, when you now change a parameter inside the paragraph its new value will not be applied (unless you prefix it with `\frozen` or `\snapshot` it) but no one did that anyway.

11.4 Wrapping up

In `ConTEXtLMTX` we have a mechanism to exercise macros (or content) before a paragraph

Although the high level interface has been around for a while it still needs a bit more testing (read: use cases are needed). In the few cases where we already use it application can be different because again it relates to snapshots. This because in the past we had to use tricks that also influenced the user interface of some macros (which made them less natural as one would expect). So the question is: where do we apply it in old mechanisms and where not.

todo: accumulation, interference, where applied, limitations

11.5 Hanging

There are two mechanisms for getting a specific paragraph shape: rectangular hanging and arbitrary shapes. Both mechanisms work top-down. The first mechanism uses a combination of `\hangafter` and `\hangindent`, and the second one depends on `\parshape`. In this section we discuss the rectangular one.

```
\hangafter 4 \hangindent 4cm \samplefile{tufte} \page
\hangafter -4 \hangindent 4cm \samplefile{tufte} \page
\hangafter 4 \hangindent -4cm \samplefile{tufte} \page
\hangafter -4 \hangindent -4cm \samplefile{tufte} \page
```

As you can see in figure 11.1, the four cases are driven by the sign of the values. If you want

11.6 Shapes

In `ConTEXt` we don't use `\parshape` a lot. It is used in for instance side floats but even there `MetaFun`, and the manual also needed an update, one of the examples in that manual that

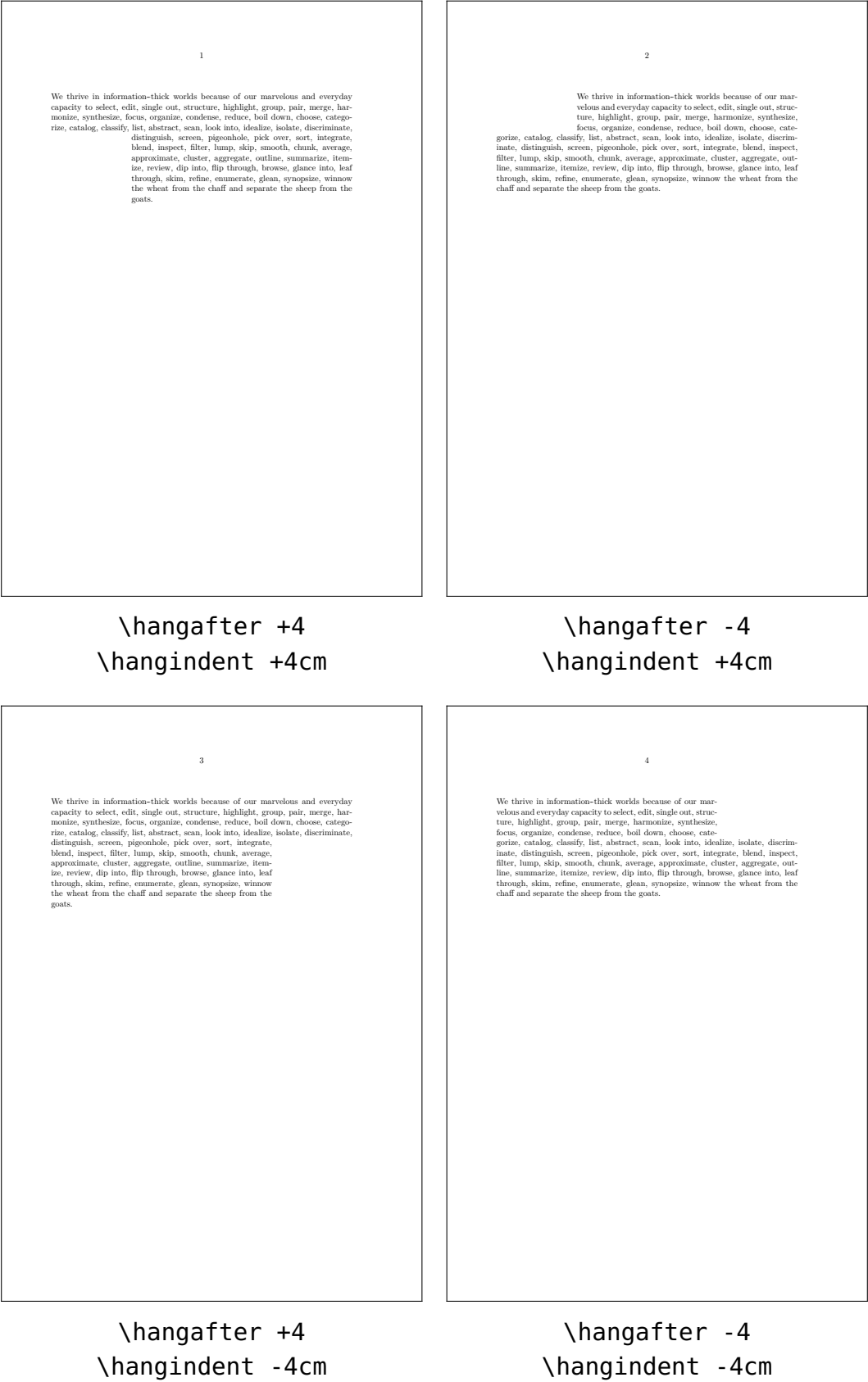


Figure 11.1 Hanging indentation

One important property of the `\parshape` mechanism is that it works per paragraph. You define a shape in terms of a left margin and width of a line. The shape has a fixed number of such pairs and when there is more content, the last one is used for the rest of the lines. When the paragraph is finished, the shape is forgotten.¹⁸

The high level interface is a follow up on the example in the MetaFun manual and uses shapes that carry over to the next paragraph. In addition we can cycle over a shape. In this interface shapes are defined using keyword. Here are some examples:

```
\startparagraphshape[test]
  left 1mm right 1mm
  left 5mm right 5mm
\stopparagraphshape
```

This shape has only two entries so the first line will have a 1mm margin while later lines will get 5mm margins. This translates into a `\parshape` like:

```
\parshape 2
  1mm \dimexpr\hsize-1mm\relax
  5mm \dimexpr\hsize-5mm\relax
```

Watch the number 2: it tells how many specification lines follow. As you see, we need to calculate the width.

```
\startparagraphshape[test]
  left 1mm right 1mm
  left 5mm right 5mm
  repeat
\stopparagraphshape
```

This variant will alternate between 1mm and 5mm margins. The repeating feature is translated as follows. Maybe at some point I will introduce a few more options.

```
\parshape 2 options 1
  1mm \dimexpr\hsize-1mm\relax
  5mm \dimexpr\hsize-5mm\relax
```

A shape can have some repetition, and we can save keystrokes by copying the last entry. The resulting `\parshape` becomes rather long.

¹⁸ Not discussed here is a variant that might end up in LuaMetaTeX that works with the progression, i.e. takes the height of the content so far into account. This is somewhat tricky because for that to work vertical skips need to be frozen, which is no real big deal but has to be done careful in the code.

```

\startparagraphshape[test]
  left 1mm right 1mm
  left 2mm right 2mm
  left 3mm right 3mm
  copy 8
  left 4mm right 4mm
  left 5mm right 5mm
  left 5mm hsize 10cm

```

```
\stopparagraphshape
```

Also watch the hsize keyword: we don't calculate the hsize from the left and right values but explicitly set it.

```

\startparagraphshape[test]
  left 1mm right 1mm
  right 3mm
  left 5mm right 5mm
  repeat

```

```
\stopparagraphshape
```

When a right keywords comes first the left is assumed to be zero. In the examples that follow we will use a couple of definitions:

```

\startparagraphshape[test]
  both 1mm both 2mm both 3mm both 4mm both 5mm both 6mm
  both 7mm both 6mm both 5mm both 4mm both 3mm both 2mm

```

```
\stopparagraphshape
```

```

\startparagraphshape[test-repeat]
  both 1mm both 2mm both 3mm both 4mm both 5mm both 6mm
  both 7mm both 6mm both 5mm both 4mm both 3mm both 2mm
  repeat

```

```
\stopparagraphshape
```

The last one could also be defines as:

```

\startparagraphshape[test-repeat]
  \rawparagraphshape{test} repeat
\stopparagraphshape

```

In the previous code we already introduced the repeat option. This will make the shape repeat at the engine level when the shape runs out of specified lines. In the application of a shape definition we can specify a method to be used and that deter-

mine_if_the_next_paragraph_will_start_where_we_left_off_and_discard_afterwards_(shift)_or_that_we_move_the_discarded_lines_up_front_so_that_we_never_run_out_of_lines_(cycle). It sounds complicated but just keep in mind that repeat is part of the \parshape and act within a paragraph while shift and cycle are applied when a new paragraph is started.

In figure 11.2 you see the following applied:

```
\startshapedparagraph[list=test]
  \dorecurse{8}{\showparagraphshape\samplefile{tufte} \par}
\stopshapedparagraph

\startshapedparagraph[list=test-repeat]
  \dorecurse{8}{\showparagraphshape\samplefile{tufte} \par}
\stopshapedparagraph
```

In figure 11.3 we use this instead:

```
\startshapedparagraph[list=test,method=shift]
  \dorecurse{8}{\showparagraphshape\samplefile{tufte} \par}
\stopshapedparagraph
```

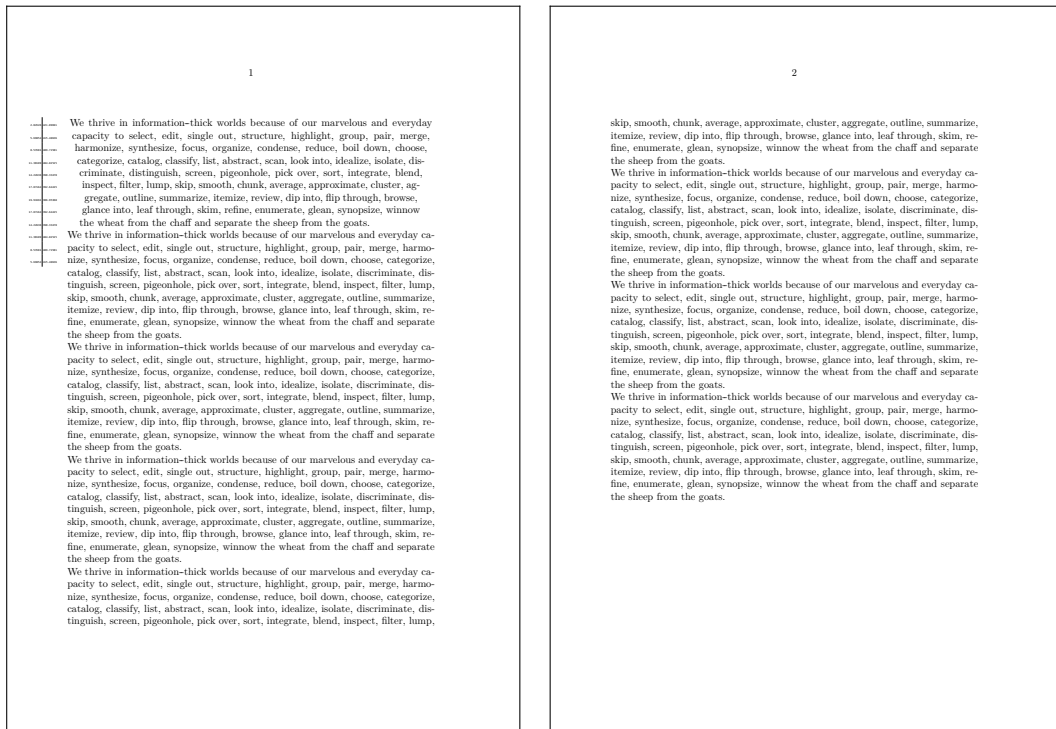
Finally, in figure 11.4 we use:

```
\startshapedparagraph[list=test,method=cycle]
  \dorecurse{8}{\showparagraphshape\samplefile{tufte} \par}
\stopshapedparagraph
```

These examples are probably too small to see the details but you can run them yourself or zoom in on the details. In the margin we show the values used. Here is a simple example of (non) poetry. There are other environments that can be used instead but this makes a good example anyway.

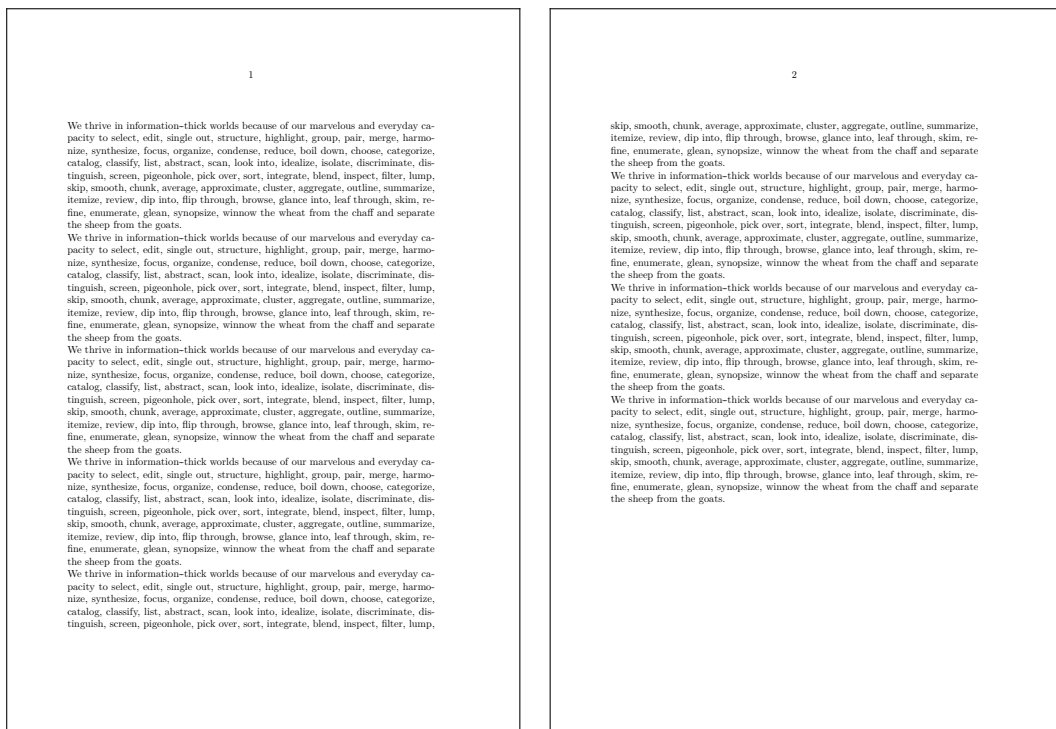
```
\startparagraphshape[test]
  left 0em right 0em
  left 1em right 0em
  repeat
\stopparagraphshape

\startshapedparagraph[list=test,method=cycle]
  verse line 1.1\crlf verse line 2.1\crlf
  verse line 3.1\crlf verse line 4.1\par
  verse line 1.2\crlf verse line 2.2\crlf
```



discard, finite shape, page 1

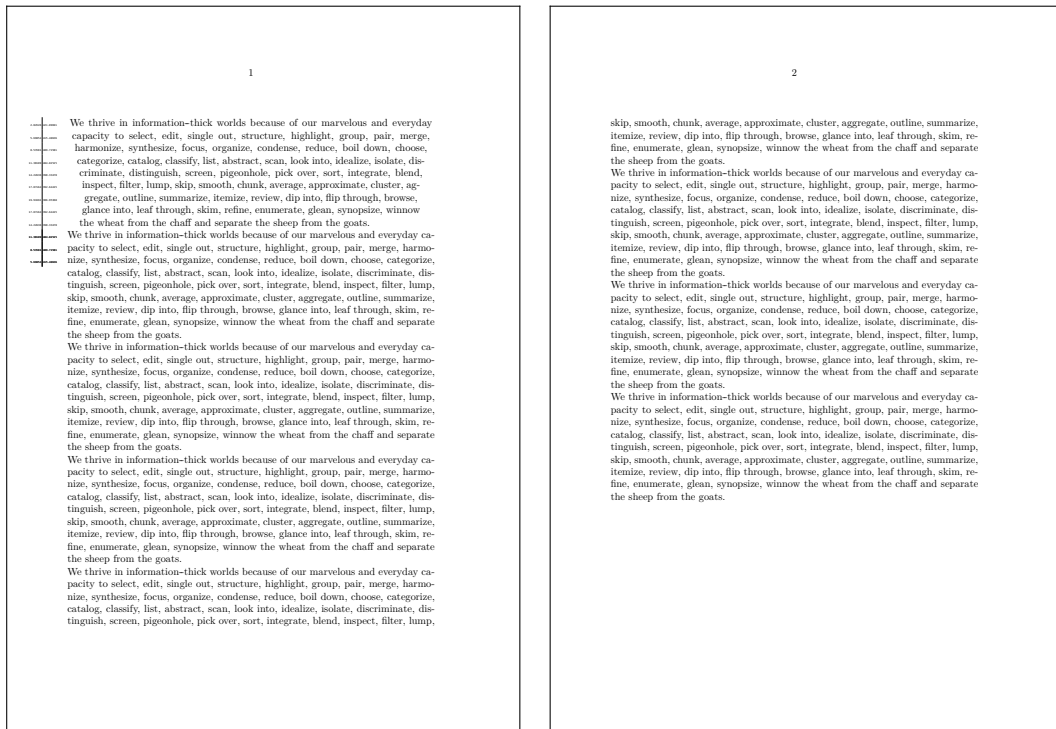
discard, finite shape, page 2



discard, repeat in shape, page 1

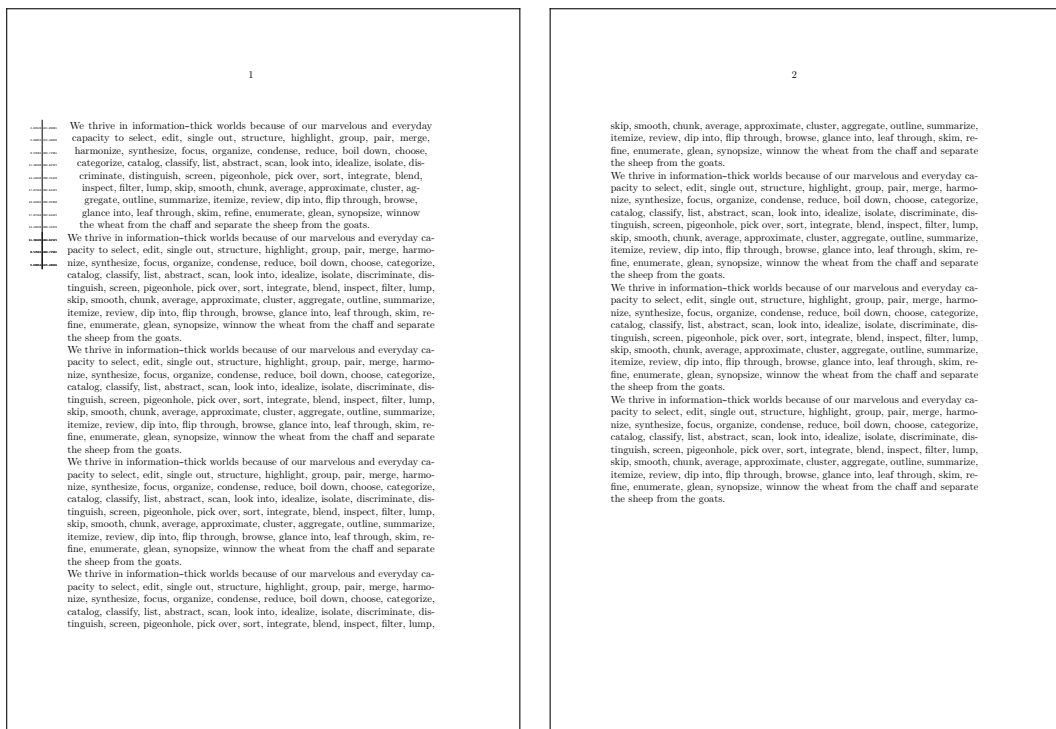
discard, repeat in shape, page 2

Figure 11.2 Discarded shaping



shift, finite shape, page 1

shift, finite shape, page 2



shift, repeat in shape, page 1

shift, repeat in shape, page 2

Figure 11.3 Shifted shaping


```

    verse_line_3.2\crlf verse_line_4.2\crlf
    verse_line_5.2\crlf verse_line_6.2\par
\stopshapedparagraph

```

```

verse_line_1.1
  verse_line_2.1
verse_line_3.1
  verse_line_4.1

```

```

verse_line_1.2
  verse_line_2.2
verse_line_3.2
  verse_line_4.2
verse_line_5.2
  verse_line_6.2

```

Because the idea for this feature originates in MetaFun, we will now kick in some MetaPost. The following code creates a shape for a circle. We use a 2mm offset here:

```

\startuseMPgraphic{circle}
  path p ; p := fullcircle scaled TextWidth ;
  build_parshape(p,
    2mm, 0, 0,
    LineHeight, StrutHeight, StrutDepth, StrutHeight
  ) ;
\stopuseMPgraphic

```

We plug this into the already described macros:

```

\startshapedparagraph[mp=circle]%
  \setupalign[verytolerant,stretch,last]%
  \samplefile{tufte}
  \samplefile{tufte}
\stopshapedparagraph

```

And get ourself a circular shape. Watch out, at this moment the shape environment does not add grouping so when as in this case you change the alignment it can influence the document.

We thrive in information-
 thick worlds because of our marvelous
 and everyday capacity to select, edit, single
 out, structure, highlight, group, pair, merge, harmo-

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

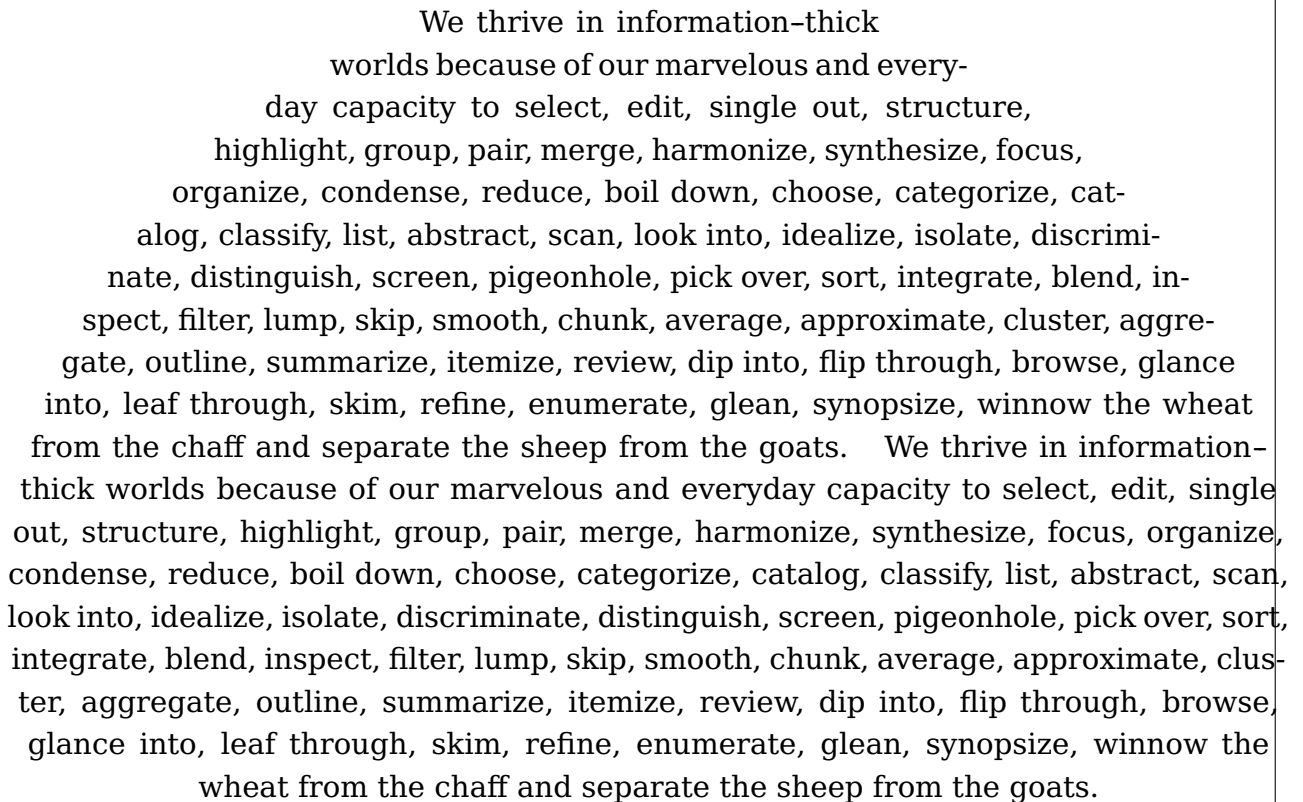
Assuming that the shape definition above is in a buffer we can do this:

```
\startshapedparagraph[mp=circle]%
  \setupalign[verytolerant,stretch,last]%
  \samplefile{tufte}
  \samplefile{tufte}
\stopshapedparagraph
```

The result is shown in figure 11.5. Because all action happens in the framed environment, v

```
\startuseMPgraphic{circle}
  path p ; p := fullcircle scaled \the\dimexpr\framedwidth+\framedoffset
    *2\relax ;
  build_parshape(p,
    \framedoffset, 0, 0,
    LineHeight, StrutHeight, StrutDepth, StrutHeight
  ) ;
  draw p ;
\stopuseMPgraphic
```

A mechanism like this is often never completely automatic in the sense that you need to keep an eye on the results. Depending on user demands more features can be added. With weird shapes you might want to set up the alignment to be tolerant and have some stretch.



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

Figure 11.5 A framed circular shape

The_interface_described_in_the_MetaFun_manual_is_pretty_old, the_time_stamp_of_the_original_code_is_mid_2000, but_the_principles_didn't_change. The_examples_in_meta-imp-txt. now_be_written_as:

```
\startshapetext[test 1,test 2,test 3,test 4]
  \setupalign[verytolerant,stretch,normal]%
  \samplefile{douglas} % Douglas R. Hofstadter
\stopshapetext
\startcombination[2*2]
  {\framed[offset=overlay,frame=off,background=test 1]{\getshapetext}}
    {test 1}
  {\framed[offset=overlay,frame=off,background=test 2]{\getshapetext}}
    {test 2}
  {\framed[offset=overlay,frame=off,background=test 3]{\getshapetext}}
    {test 3}
  {\framed[offset=overlay,frame=off,background=test 4]{\getshapetext}}
    {test 4}
\stopcombination
```

In figure 11.6 we see the result. Watch how for two shapes we have enabled tracing. Of co

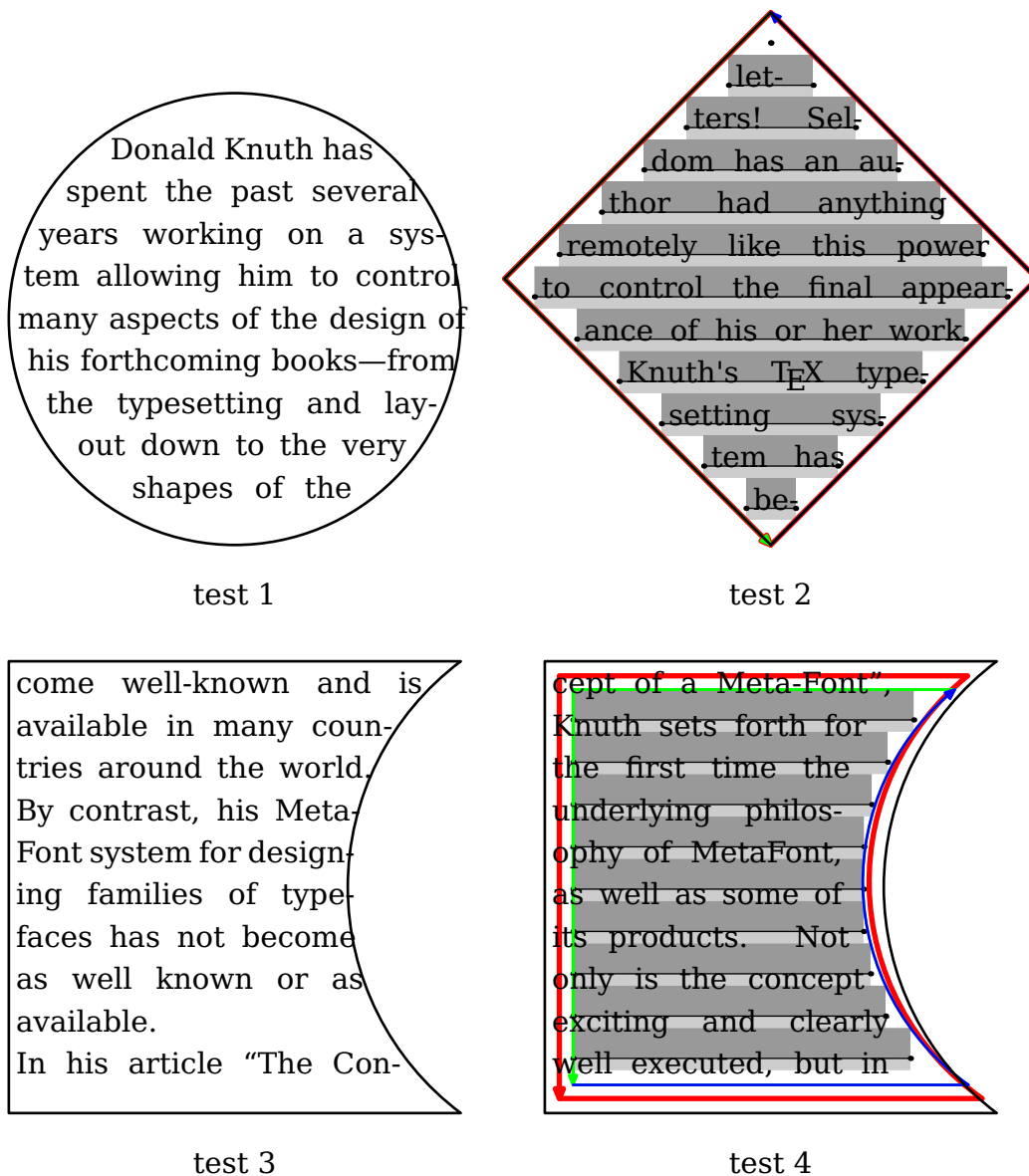


Figure 11.6

Here is a bit more extreme example. Again we use a circle:

```
\startuseMPgraphic{circle}
  lmt_parshape [
    path      = fullcircle scaled 136mm,
    offset    = 2mm,
    bottomskip = - 1.5LineHeight,
  ] ;
\stopuseMPgraphic
```

But_we_output_a_longer_text:_

```
\startshapedparagraph[mp=circle,repeat=yes,method=cycle]%
  \setupalign[verytolerant,stretch,last]\dontcomplain
  {\darkred      \samplefile{tufte}}\par
  {\darkgreen    \samplefile{tufte}}\par
  {\darkblue     \samplefile{tufte}}\par
  {\darkcyan     \samplefile{tufte}}\par
  {\darkmagenta  \samplefile{tufte}}\par
\stopshapedparagraph
```

We_get_a_multi-page_shape:_

We_thrive_in_information-
 thick_worlds_because_of_our_marvelous_
 and_everyday_capacity_to_select,_edit,_single_
 out,_structure,_highlight,_group,_pair,_merge,_harmoni-
 zize,_synthesize,_focus,_organize,_condense,_reduce,_boil_
 down,_choose,_categorize,_catalog,_classify,_list,_abstract,_scan,_look_
 into,_idealize,_isolate,_discriminate,_distinguish,_screen,_pigeonhole,_pick_
 over,_sort,_integrate,_blend,_inspect,_filter,_lump,_skip,_smooth,_chunk,_av-
 erage,_approximate,_cluster,_aggregate,_outline,_summarize,_itemize,_re-
 view,_dip_into,_flip_through,_browse,_glance_into,_leaf_through,_skim,_re-
 fine,_enumerate,_glean,_synopsise,_winnow_the_wheat_from_the_chaff_and_sepa-
 rate_the_sheep_from_the_goats._

We_thrive_in_information-thick_worlds_because_of_our_marvelous_and_everyday_
 capacity_to_select,_edit,_single_out,_structure,_highlight,_group,_pair,_merge,_har-
 monize,_synthesize,_focus,_organize,_condense,_reduce,_boil_down,_choose,_cat-
 egorize,_catalog,_classify,_list,_abstract,_scan,_look_into,_idealize,_isolate,_dis-
 criminate,_distinguish,_screen,_pigeonhole,_pick_over,_sort,_integrate,_blend,_in-
 spect,_filter,_lump,_skip,_smooth,_chunk,_average,_approximate,_cluster,_ag-
 gregate,_outline,_summarize,_itemize,_review,_dip_into,_flip_through,_browse,_glance_
 into,_leaf_through,_skim,_refine,_enumerate,_glean,_synopsise,_winnow_the_
 wheat_from_the_chaff_and_separate_the_sheep_from_the_goats._

We_thrive_in_information-thick_worlds_because_of_our_marvelous_
 and_everyday_capacity_to_select,_edit,_single_out,_structure,_high-
 light,_group,_pair,_merge,_harmonize,_synthesize,_focus,_or-
 ganize,_condense,_reduce,_boil_down,_choose,_catego-
 rize,_catalog,_classify,_list,_abstract,_scan,_look_
 into,_idealize,_isolate,_discriminate,_distin-

guish, _screen, _pigeonhole, _pick_
 over, _sort, _in-
 tegrate, _blend, _inspect, _fil-
 ter, _lump, _skip, _smooth, _chunk, _av-
 erage, _approximate, _cluster, _aggregate, _out-
 line, _summarize, _itemize, _review, _dip_into, _flip_through, _browse, _glance_
 into, _leaf_through, _skim, _refine, _enumerate, _glean, _synop-
 size, _winnow_the_wheat_from_the_chaff_and_separate_the_sheep_
 from_the_goats._

We_thrive_in_information-thick_worlds_because_of_our_marvelous_and_
 everyday_capacity_to_select, _edit, _single_out, _structure, _highlight, _group, _pair, _merge, _har-
 monize, _synthesize, _focus, _organize, _condense, _reduce, _boil_down, _choose, _cat-
 egorize, _catalog, _classify, _list, _abstract, _scan, _look_into, _idealize, _isolate, _dis-
 criminate, _distinguish, _screen, _pigeonhole, _pick_over, _sort, _integrate, _blend, _in-
 spect, _filter, _lump, _skip, _smooth, _chunk, _average, _approximate, _cluster, _aggre-
 gate, _outline, _summarize, _itemize, _review, _dip_into, _flip_through, _browse, _glance_
 into, _leaf_through, _skim, _refine, _enumerate, _glean, _synopsi-
 size, _winnow_the_wheat_from_the_chaff_and_separate_the_sheep_from_the_goats._

We_thrive_in_information-thick_worlds_because_of_our_marvelous_and_everyday_
 capacity_to_select, _edit, _single_out, _structure, _highlight, _group, _pair, _merge, _har-
 monize, _synthesize, _focus, _organize, _condense, _reduce, _boil_down, _choose, _cat-
 egorize, _catalog, _classify, _list, _abstract, _scan, _look_into, _idealize, _isolate, _dis-
 criminate, _distinguish, _screen, _pigeonhole, _pick_over, _sort, _integrate, _blend, _in-
 spect, _filter, _lump, _skip, _smooth, _chunk, _average, _approximate, _clus-
 ter, _aggregate, _outline, _summarize, _itemize, _review, _dip_into, _flip_
 through, _browse, _glance_into, _leaf_through, _skim, _refine, _enu-
 merate, _glean, _synopsi-
 size, _winnow_the_wheat_from_the_
 chaff_and_separate_the_sheep_from_the_goats._

Compare_this_with:_

```
\startshapedparagraph[mp=circle,repeat=yes,method=cycle]%
  \setupalign[verytolerant,stretch,last]\dontcomplain
  {\darkred      \samplefile{tufte}}
  {\darkgreen    \samplefile{tufte}}
  {\darkblue     \samplefile{tufte}}
  {\darkcyan     \samplefile{tufte}}
  {\darkmagenta  \samplefile{tufte}}
\stopshapedparagraph
```

Which_gives:_

We_thrive_in_information-
 thick_worlds_because_of_our_marvelous_
 and_everyday_capacity_to_select,_edit,_single_
 out,_structure,_highlight,_group,_pair,_merge,_harmo-
 nize,_synthesize,_focus,_organize,_condense,_reduce,_boil_
 down,_choose,_categorize,_catalog,_classify,_list,_abstract,_scan,_look_
 into,_idealize,_isolate,_discriminate,_distinguish,_screen,_pigeonhole,_pick_
 over,_sort,_integrate,_blend,_inspect,_filter,_lump,_skip,_smooth,_chunk,_av-
 erage,_approximate,_cluster,_aggregate,_outline,_summarize,_itemize,_re-
 view,_dip_into,_flip_through,_browse,_glance_into,_leaf_through,_skim,_re-
 fine,_enumerate,_glean,_synopsise,_winnow_the_wheat_from_the_chaff_and_sep-
 arate_the_sheep_from_the_goats. We_thrive_in_information-thick_worlds_be-
 cause_of_our_marvelous_and_everyday_capacity_to_select,_edit,_single_out,_struc-
 ture,_highlight,_group,_pair,_merge,_harmonize,_synthesize,_focus,_organize,_con-
 dense,_reduce,_boil_down,_choose,_categorize,_catalog,_classify,_list,_abstract,_scan,_look_
 into,_idealize,_isolate,_discriminate,_distinguish,_screen,_pigeonhole,_pick_over,_sort,_in-
 tegrate,_blend,_inspect,_filter,_lump,_skip,_smooth,_chunk,_average,_approximate,_clus-
 ter,_aggregate,_outline,_summarize,_itemize,_review,_dip_into,_flip_through,_browse,_glance_
 into,_leaf_through,_skim,_refine,_enumerate,_glean,_synopsise,_winnow_the_
 wheat_from_the_chaff_and_separate_the_sheep_from_the_goats. We_thrive_
 in_information-thick_worlds_because_of_our_marvelous_and_everyday_
 capacity_to_select,_edit,_single_out,_structure,_highlight,_group,_pair,_merge,_har-
 monize,_synthesize,_focus,_organize,_condense,_reduce,_boil_down,_choose,_cat-
 egorize,_catalog,_classify,_list,_abstract,_scan,_look_into,_ide-
 alize,_isolate,_discriminate,_distinguish,_screen,_pigeon-
 hole,_pick_over,_sort,_integrate,_blend,_inspect,_fil-
 ter,_lump,_skip,_smooth,_chunk,_average,_ap-
 proximate,_cluster,_aggregate,_out-
 line,_summa-
 rize,_itemize,_review,_dip_
 into,_flip_through,_browse,_glance_into,_leaf_
 through,_skim,_refine,_enumerate,_glean,_synop-
 size,_winnow_the_wheat_from_the_chaff_and_separate_
 the_sheep_from_the_goats. We_thrive_in_information-thick_
 worlds_because_of_our_marvelous_and_everyday_capacity_to_se-
 lect,_edit,_single_out,_structure,_highlight,_group,_pair,_merge,_har-
 monize,_synthesize,_focus,_organize,_condense,_reduce,_boil_down,_choose,_cat-
 egorize,_catalog,_classify,_list,_abstract,_scan,_look_into,_idealize,_isolate,_dis-
 criminate,_distinguish,_screen,_pigeonhole,_pick_over,_sort,_integrate,_blend,_in-

spect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip_into, flip_through, browse, glance_into, leaf_through, skim, refine, enumerate, glean, synopsise, winnow_the_wheat_from_the_chaff_and_separate_the_sheep_from_the_goats. We thrive in information-thick worlds because of our marvelous and everyday capacity to select, edit, single_out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil_down, choose, categorize, catalog, classify, list, abstract, scan, look_into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick_over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip_into, flip_through, browse, glance_into, leaf_through, skim, refine, enumerate, glean, synopsise, winnow_the_wheat_from_the_chaff_and_separate_the_sheep_from_the_goats.

Here the `bottomskip` takes care of subtle rounding issues as well as discarding the last line in the shape so that we get nicer continuation. There is no full automated solution for all you can come up with.

Mixing a `MetaPost` specification into a regular one is also possible. The next example demonstrates this as well as the option to remove some lines from a specification:

```
\startparagraphshape[test]
  left 0em right 0em
  left 1em right 0em
  metapost {circle}
  delete 3
  metapost {circle,circle,circle}
  delete 7
  metapost {circle}
  repeat
\stopparagraphshape
```

You can combine a shape with narrowing a paragraph. Watch the `absolute` keyword in the next code. The result is shown in figure 11.7.

```
\startuseMPgraphic{circle}
  lmt_parshape [
    path      = fullcircle scaled TextWidth,
    bottomskip = - 1.5LineHeight,
  ] ;
\stopuseMPgraphic
```

```

\startparagraphshape[test-1]
  metapost {circle} repeat
\stopparagraphshape

\startparagraphshape[test-2]
  absolute left metapost {circle} repeat
\stopparagraphshape

\startparagraphshape[test-3]
  absolute right metapost {circle} repeat
\stopparagraphshape

\startparagraphshape[test-4]
  absolute both metapost {circle} repeat
\stopparagraphshape

\showframe

\startnarrower[4*left,2*right]
  \startshapedparagraph[list=test-1,repeat=yes,method=repeat]%
    \setupalign[verytolerant,stretch,last]\dontcomplain
    \dorecurse{3}{\samplefile{thuan}}
  \stopshapedparagraph
  \page
  \startshapedparagraph[list=test-2,repeat=yes,method=repeat]%
    \setupalign[verytolerant,stretch,last]\dontcomplain
    \dorecurse{3}{\samplefile{thuan}}
  \stopshapedparagraph
  \page
  \startshapedparagraph[list=test-3,repeat=yes,method=repeat]%
    \setupalign[verytolerant,stretch,last]\dontcomplain
    \dorecurse{3}{\samplefile{thuan}}
  \stopshapedparagraph
  \page
  \startshapedparagraph[list=test-4,repeat=yes,method=repeat]%
    \setupalign[verytolerant,stretch,last]\dontcomplain
    \dorecurse{3}{\samplefile{thuan}}
  \stopshapedparagraph
\stopnarrower

```

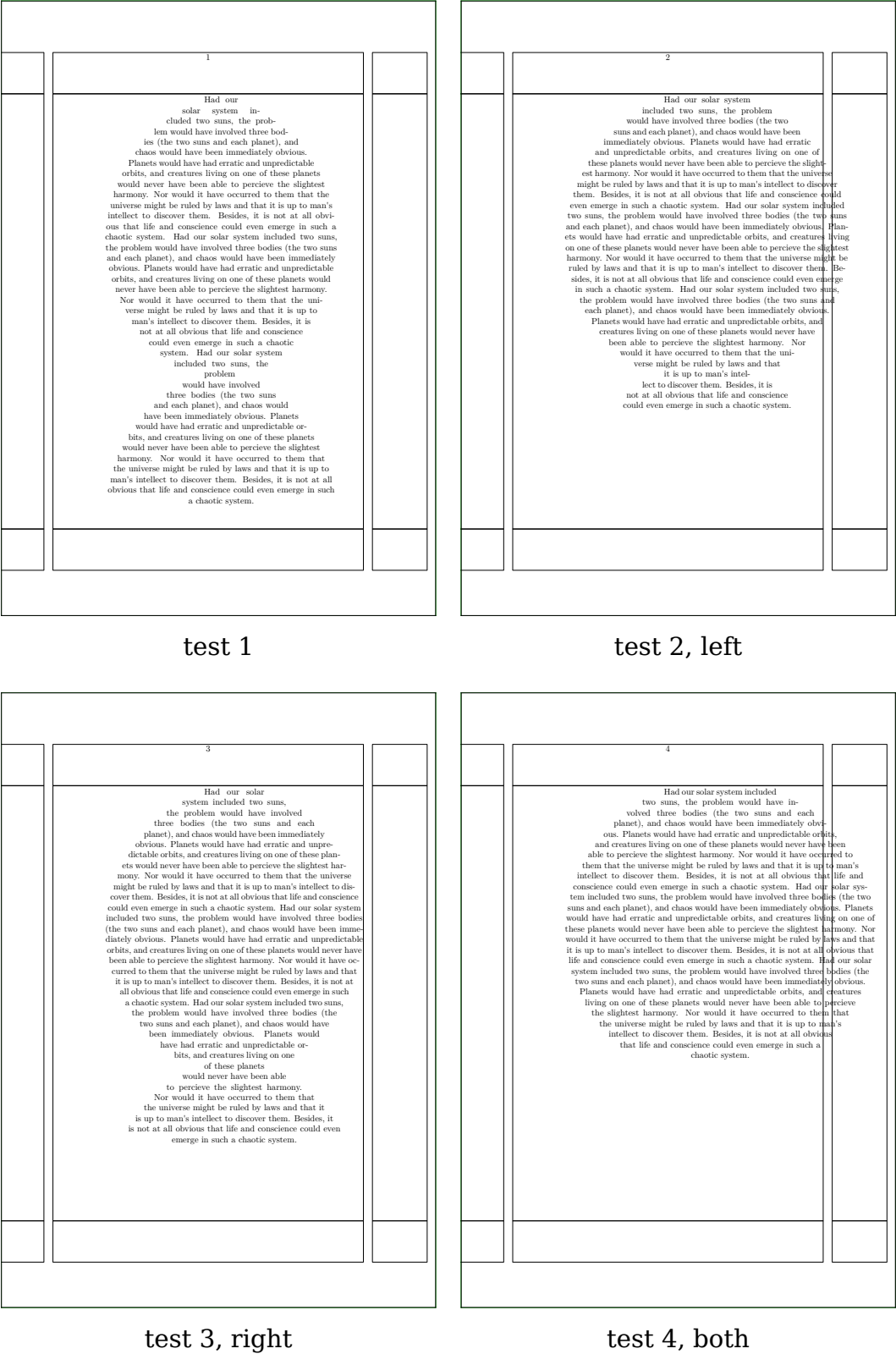


Figure 11.7 Skip compensation

The `shape` mechanism has a few more tricks but these are really meant for usage in specific situations, where one knows what one deals with. The following examples are visualized in figure 11.8.

```
\useMPlibrary[dum]
\usemodule[article-basics]

\startbuffer
  \externalfigure[dummy][width=6cm]
\stopbuffer

\startshapedparagraph[text=\getbuffer]
  \dorecurse{3}{\samplefile{ward}\par}
\stopshapedparagraph

\page

\startshapedparagraph[text=\getbuffer,distance=1em]
  \dorecurse{3}{\samplefile{ward}\par}
\stopshapedparagraph

\page

\startshapedparagraph[text=\getbuffer,distance=1em,
  hoffset=-2em]
  \dorecurse{3}{\samplefile{ward}\par}
\stopshapedparagraph

\page

\startshapedparagraph[text=\getbuffer,distance=1em,
  voffset=-2ex,hoffset=-2em]
  \dorecurse{3}{\samplefile{ward}\par}
\stopshapedparagraph

\page

\startshapedparagraph[text=\getbuffer,distance=1em,
  voffset=-2ex,hoffset=-2em,lines=1]
  \dorecurse{3}{\samplefile{ward}\par}
\stopshapedparagraph
```

\page

\startshapedparagraph[width=4cm,lines=4]

\dorecurse{3}{\samplefile{ward}\par}

\stopshapedparagraph



Figure 11.8 Flow around something

11.7 Modes

todo: some of the side effects of so called modes

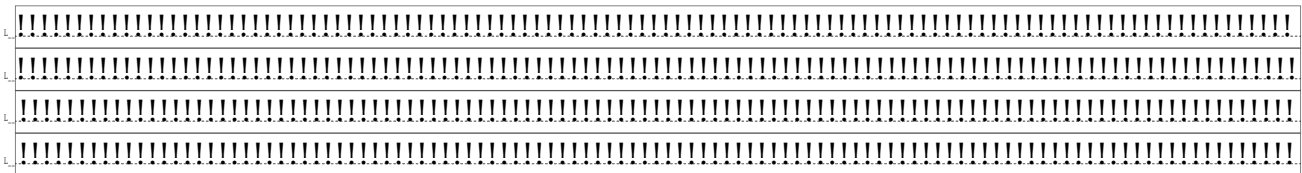
Modes

11.8 Leaders

Leaders are a basic feature that users probably never run into directly. They repeat content till it fits the specified width which can be stretched out. The content is typeset once and it is the backend that does the real work of repetition.

```
\strut\leaders \hbox{!}\hfill\strut
\strut\xleaders\hbox{!}\hfill\strut
\strut\cleaders\hbox{!}\hfill\strut
\strut\gleaders\hbox{!}\hfill\strut
```

Here `\leaders` starts at the left edge and repeats the box as long as it fits, `\xleaders` till the edges and `\cleaders` centers the lot. The `\gleaders` primitive (which is not in original \TeX) takes the outer box as reference and further behaves like `\cleaders`.

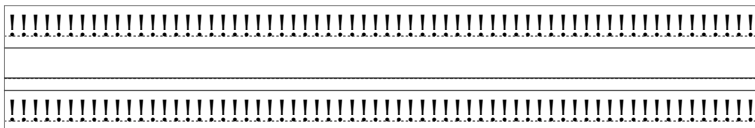


The leader primitives take box or rule but in LuaMeta \TeX a glyph can also be specified, where

```
\ruledvbox \bgroup \hsize 10cm
  \strut\cleaders\hbox{!}\hfill\strut
\egroup
```

```
\ruledvbox \bgroup \hsize 10cm
  \strut\cleaders\hrule\hfill\strut
\egroup
```

```
\ruledvbox \bgroup \hsize 10cm
  \strut\cleaders\glyph`!\hfill\strut
\egroup
```



The LuaMeta \TeX engine also introduced `\uleaders`

We show three boxes, a regular one first (red):

```
x xx xxx xxxx
```

```
\ruledhbox{L\hss R}\space
x xx xxx xxxx
```

The second one (blue) is also a box but one that stretches upto 100pt and is in a later stage, when the paragraph has been built, is repackaged to the effective width. The third example (green) leaves out the background.

```
x xx xxx xxxx L R x xx xxx xxxx x xx xxx xxxx L R x xx xxx xxxx x xx xxx xxxx
x xx xxx xxxx x xx xxx xxxx L R x xx xxx xxxx x xx xxx xxxx L R x xx xxx xxxx x xx xxx xxxx
x xx xxx xxxx x xx xxx xxxx L R x xx xxx xxxx x xx xxx xxxx L R x xx xxx xxxx x xx xxx xxxx
xxxxxxx[R]xx xxxxxxxxIx x Rxxxxxxx[R]xxxxxxxLxx Rxxxxxxx[R]xxxxxxx xxxxxxx [L]
x xx xxx xxxx x xx xxx xxxx L R x xx xxx xxxx x xx xxx xxxx L R x xx xxx xxxx x xx xxx xxxx
x xx xxx xxxx x xx xxx xxxx L R x xx xxx xxxx x xx xxx xxxx L R x xx xxx xxxx x xx xxx xxxx
x xx xxx xxxx x xx xxx xxxx LR x xx xxx xxxx x xx xxx xxxx LR x xx xxx xxxx
```

In ConT_EXt we have wrapped this feature in the adaptive box mechanism, so here a few a f

```
\startsetups adaptive:test:a
  \setbox\usedadaptivebox\vbox to \usedadaptivetotal \bgroup
  \externalfigure
    [cow.pdf]
    [width=\framedmaxwidth,
     frame=on,
     height=\usedadaptivetotal]%
  \egroup
\stopsetups
```

```
\startsetups adaptive:test:b
  \setbox\usedadaptivebox\vbox to \usedadaptivetotal \bgroup
  \externalfigure
    [cow.pdf]
    [width=\usedadaptivewidth,
     frame=on,
     height=\usedadaptivetotal]%
  \egroup
\stopsetups
```

We use this as follows (see figure 11.9 for the result):

```
\framed[height=18cm,align=middle,adaptive=yes,top=,bottom=] {%
  \begstrut \samplefile{tufte} \endstrut
  \par
```

```

\adaptivevbox
  [strut=yes, setups=adaptive:test:a]
  {\showstruts\strut\hsize5cm\hss}%
\par
\adaptivevbox
  [strut=yes, setups=adaptive:test:b]
  {\showstruts\strut\hsize5cm\hss}%
\par
\beginstrut \samplefile{tufte} \endstrut
}

```

Here is one that you can test yourself:

```

\startsetups adaptive:test
  \setbox\usedadaptivebox\vbox to \usedadaptivetotal \bgroup
  \externalfigure
    [cow.pdf]
    [width=\usedadaptivewidth,
     height=\usedadaptivetotal]%
\egroup
\stopsetups

\ruledvbox to \textheight {
  \par \beginstrut \samplefile{tufte} \endstrut \par
  \adaptivevbox[strut=yes, setups=adaptive:test]{\hsize\textwidth\hss}
  \par \beginstrut \samplefile{tufte} \endstrut
}

```

The next example comes from the test suite (where it runs over many pages in order to illustrate the idea):

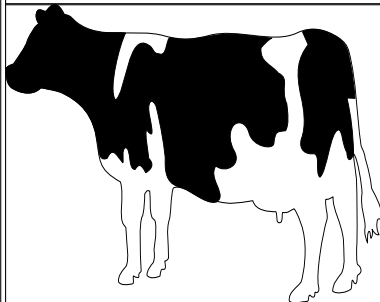
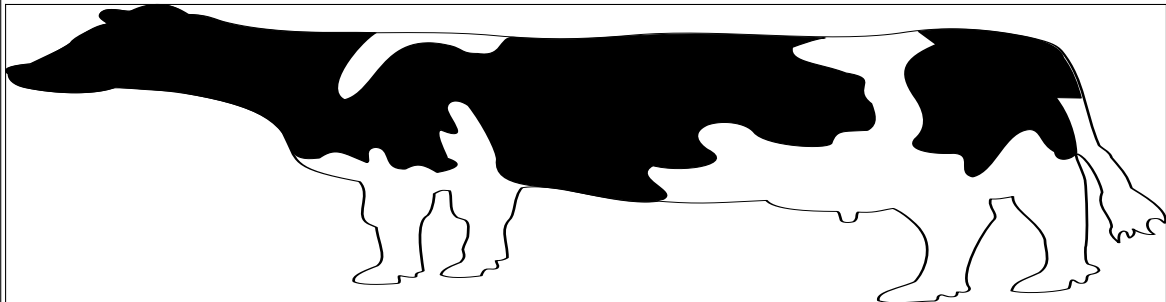
```

\startMPdefinitions
  def TickTock =
    interim linecap := squared;
    save p ; path p ;
    p := fullsquare xysized(AdaptiveWidth,.9(AdaptiveHeight+AdaptiveDepth))
;

  fill p withcolor AdaptiveColor ;
  draw bottomboundary (p enlarged (-AdaptiveThickness) )
    withdashes (3*AdaptiveThickness)
    withpen pencircle scaled AdaptiveThickness

```

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



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

Figure 11.9

```

        withcolor white ;
    enddef ;
\stopMPdefinitions

\startsetups adaptive:test
    \setbox\usedadaptivebox\hbox

```

```

        to          \usedadaptivewidth
        yoffset -.9\usedadaptivedepth
\bgroup
    \hss
    \startMPcode
        TickTock ;
    \stopMPcode
    \hss
\egroup
\stopsetups

```

```

\definecolor[adaptive:tick][.25(blue,green)]
\definecolor[adaptive:tock][.75(blue,green)]

```

```

\defineadaptive
[tick]
[setups=adaptive:test,
color=adaptive:tick,
foregroundcolor=white,
foregroundstyle=\infofont,
strut=yes]

```

```

\defineadaptive
[tock]
[tick]
[color=adaptive:tock]

```

```
































\dostepwiserecurse{8}{12}{1}{%
    \dostepwiserecurse{5}{15}{1}{%
        this~#1.##1 is~#1.##1 test~#1.##1
        \ifodd##1\relax
            \adaptivebox[tick]{\hss tick #1.##1\hss}
        \else
            \adaptivebox[tock]{\hss tock #1.##1\hss}
        \fi
    }
}

```

```

this 8.5_is 8.5_test 8.5_ tick 8.5 this 8.6_is 8.6_test 8.6_ tock 8.6 this 8.7_is 8.7_test 8.7_
_this 8.8_is 8.8_test 8.8_ tock 8.8 this 8.9_is 8.9_test 8.9_ tick 8.9 this 8.10_is 8.10_test 8.10_
_this 8.11_is 8.11_test 8.11_ tick 8.11 this 8.12_is 8.12_test 8.12_ tock 8.12 this 8.13_is 8.13_test 8.13_

```

_this 8.14_is 8.14_test 8.14_  _this 8.15_is 8.15_test 8.15_  _this 9.5_is 9.5_test 9.5_
 _this 9.6_is 9.6_test 9.6_  _this 9.7_is 9.7_test 9.7_  _this 9.8_is 9.8_test 9.8_
 _this 9.9_is 9.9_test 9.9_  _this 9.10_is 9.10_test 9.10_  _this 9.11_is 9.11_test 9.11_
 _this 9.12_is 9.12_test 9.12_  _this 9.13_is 9.13_test 9.13_  _this 9.14_is 9.14_test 9.14_
 _this 9.15_is 9.15_test 9.15_  _this 10.5_is 10.5_test 10.5_  _this 10.6_is 10.6_test 10.6_
 _this 10.7_is 10.7_test 10.7_  _this 10.8_is 10.8_test 10.8_  _this 10.9_is 10.9_test 10.9_
 _this 10.10_is 10.10_test 10.10_  _this 10.11_is 10.11_test 10.11_  _this 10.12_is 10.12_t
 _this 10.13_is 10.13_test 10.13_  _this 10.14_is 10.14_test 10.14_  _this 10.15_is 10.15_t
 _this 11.5_is 11.5_test 11.5_  _this 11.6_is 11.6_test 11.6_  _this 11.7_is 11.7_test 11.7_
 _this 11.8_is 11.8_test 11.8_  _this 11.9_is 11.9_test 11.9_  _this 11.10_is 11.10_test 11.10_
 _this 11.11_is 11.11_test 11.11_  _this 11.12_is 11.12_test 11.12_  _this 11.13_is 11.13_t
 _this 11.14_is 11.14_test 11.14_  _this 11.15_is 11.15_test 11.15_  _this 12.5_is 12.5_tes
 _this 12.6_is 12.6_test 12.6_  _this 12.7_is 12.7_test 12.7_  _this 12.8_is 12.8_test 12.8_
 _this 12.9_is 12.9_test 12.9_  _this 12.10_is 12.10_test 12.10_  _this 12.11_is 12.11_test 1
 _this 12.12_is 12.12_test 12.12_  _this 12.13_is 12.13_test 12.13_  _this 12.14_is 12.14_t
 _this 12.15_is 12.15_test 12.15_  _____














In_the_next_example_the_graphics_adapt_to_the_available_space:







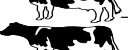
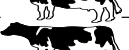








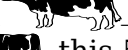

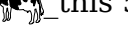



```

\startsetups adaptive:test
  \setbox\usedadaptivebox\hbox
    to      \usedadaptivewidth
    yoffset -\usedadaptivedepth
  \bgroup
    \externalfigure
      [cow.pdf]
      [width=\usedadaptivewidth,
        height=\dimexpr\usedadaptivetotal\relax]%
  \egroup
\stopsetups

\dostepwiserecurse{1}{50}{1}{%
  this~#1 is~#1 test~#1
  {\adaptivebox[strut=yes,setup=adaptive:test]}{}}
}

```

this 1_is 1_test 1_  this 2_is 2_test 2_  this 3_is 3_test 3_  this 4_is 4_test 4_
 _this 5_is 5_test 5_  _this 6_is 6_test 6_  _this 7_is 7_test 7_  _this 8_is 8_test 8_
 _this 9_is 9_test 9_  _this 10_is 10_test 10_  _this 11_is 11_test 11_  _this 12_is 12_test 12_
 _this 13_is 13_test 13_  _this 14_is 14_test 14_  _this 15_is 15_test 15_
 _this 16_is 16_test 16_  _this 17_is 17_test 17_  _this 18_is 18_test 18_

_this 19_is 19_test 19_  _this 20_is 20_test 20_  _this 21_is 21_test 21_
 _this 22_is 22_test 22_  _this 23_is 23_test 23_  _this 24_is 24_test 24_
 _this 25_is 25_test 25_  _this 26_is 26_test 26_  _this 27_is 27_test 27_
 _this 28_is 28_test 28_  _this 29_is 29_test 29_  _this 30_is 30_test 30_
 _this 31_is 31_test 31_  _this 32_is 32_test 32_  _this 33_is 33_test 33_
 _this 34_is 34_test 34_  _this 35_is 35_test 35_  _this 36_is 36_test 36_
 _this 37_is 37_test 37_  _this 38_is 38_test 38_  _this 39_is 39_test 39_
 _this 40_is 40_test 40_  _this 41_is 41_test 41_  _this 42_is 42_test 42_
 _this 43_is 43_test 43_  _this 44_is 44_test 44_  _this 45_is 45_test 45_
 _this 46_is 46_test 46_  _this 47_is 47_test 47_  _this 48_is 48_test 48_
 _this 49_is 49_test 49_  _this 50_is 50_test 50_  _

11.9 Prevdepth

The `_depth_of_a_box_is_normally_positive_but_rules_can_have_a_negative_depth_in_order_to_get_a_rule_above_the_baseline`. When `TEX` was written the assumption was that a negative depth was not possible in `MetaTEX`. However, as dealing with the property is somewhat special in the engine you should be aware of the following:

```
line 1\par line 2 \par \nointerlineskip line 3 \par
```

Assuming that we haven't set any inter paragraph spacing this gives:

line 1
line 2
line 3

Here `\nointerlineskip` is (normally) defined as:

```
\prevdepth-1000pt
```

although in `ConTEXt` we use `\ignoredepthcriterion` instead of the hard coded dimension. We can test this with the following:

```

\ruledhbox \bgroup
  \PrevTest{-10.0pt}\quad
  \PrevTest{-20.0pt}\quad
  \PrevTest{-49.9pt}\quad
  \PrevTest{-50.0pt}\quad
  \PrevTest{-50.1pt}\quad
  \PrevTest{-60.0pt}\quad
  \PrevTest{-80.0pt}%
\egroup

```

In this example we set `\ignoredepthcriterion` to `-50.0pt` instead of the normal `-1000pt`. The helper is defined as:

```
\def\PrevTest#1%
{\setbox0\ruledhbox{\strut$\textcolor{blue}{t}\textcolor{red}{f}\textcolor{blue}{#1}$}%
\dp0=#1
\vbox\bgroupt\hsz4em
FIRST\par
\unhbox0\par
LAST\par
\egroup}
```

or

```
\def\PrevTest#1%  
{\setbox0\rule{dh}{box}\strut$\text{f}#1$}%  
\dp0=#1  
\vbox\bgroup  
  \rule{dh}{FIRST}\par  
  \box0\par  
  \rule{dh}{LAST}\par  
\egroup}
```

The result is shown in figures 11.10 and 11.11. The first case is what we normally have in

FIRST	FIRST	FIRST	FIRST	FIRST	FIRST	FIRST
-10.0pt	-20.0pt	-49.9pt	-50.0pt	-50.1pt	-60.0pt	-80.0pt
LAST	LAST	LAST	LAST	LAST	LAST	LAST

Figure 11.10

FIRST	FIRST	FIRST				
-10.0pt	-20.0pt	-49.9pt				
LAST	LAST	LAST	LAST	LAST	LAST	LAST

FIRST **FIRST** **FIRST** **FIRST**
 -50.0pt -50.1pt -60.0pt -80.0pt

Figure 11.11

I'm sure one can use this effect otherwise than intended but I doubt is any user is willing to do this but the fact that we can lower the criterion makes for nice experiments. Just for the record, in figure 11.12 you see what we get with positive values:

```

\ruledhbox \bgroup
  \PrevTest{10.0pt}\quad
  \PrevTest{20.0pt}\quad
  \PrevTest{49.9pt}\quad
  \PrevTest{50.0pt}\quad
  \PrevTest{50.1pt}\quad
  \PrevTest{60.0pt}\quad
  \PrevTest{80.0pt}%
\egroup

```

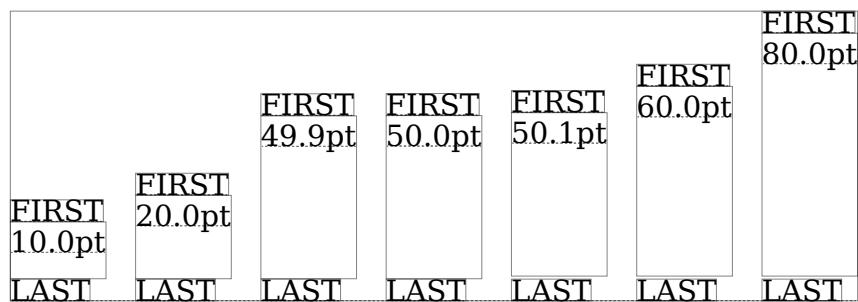


Figure 11.12

Watch_the_interline_skip_kicking_in_when_we_make_the_depth_larger_than_in_ignore-depthcriterion_being_50pt._

11.10 Normalization

todo: _users_don't_need_to_bother_about_this_but_it_might_be_interesting_anyway_

11.11 Dirty tricks

todo: _explain_example_for_combining_paragraphs_

11.12 Penalties

In_figure_11.13_we_demonstrate_the_(accumulated)_effect_of_a_few_penalty_arrays_that_you_c

```

\interlinepenalty 0
\clubpenalty      0
\widowpenalty     0
\orphanpenalty    0
\shapingpenalty   0

```

```

\clubpenalties      5    1000    2000    3000    4000    5000 % 6 -> 0
\widowpenalties     5      10      20      30      40      50 % 6 -> 0
\orphanpenalties    5       1       2       3       4       5 % 6 -> 0
\interlinepenalties 5 100000 200000 300000 400000 500000 % 6 -> 0

```

It actually makes sense to explicitly zero the last entry because as you can see in the figure the last one gets used when we run out of entries.

Can you guess what the next specification does?

```

\widowpenalties 3 options \largestspecificationoptioncode 3000 2000 1000
\clubpenalties  3 options \largestspecificationoptioncode   30   20   10

```

11.13 Par passes

Everything comes together in what we call par passes. Before we explain them first something about a feature that makes setting up for instance `\widowpenalties` easier. Here are a few definitions:

```

\specificationdef\strictwidowpenalties      \widowpenalties \plusthree
    \maxcount \maxcount \zerocount \relax
\specificationdef\strictwidowpenaltiestwo    \widowpenalties \plustwo
    \maxcount \zerocount \relax
\specificationdef\strictwidowpenaltiesthree  \widowpenalties \plusthree
    \maxcount \maxcount \zerocount \relax
\specificationdef\strictwidowpenaltiesfour   \widowpenalties \plusfour
    \maxcount \maxcount \maxcount \zerocount \relax

```

These are defined in the core and hooked into the alignment interface:

```

\installaligncommand{strictwidows} {\strictwidowpenalties }
\installaligncommand{strictwidows:2}{\strictwidowpenaltiestwo }
\installaligncommand{strictwidows:3}{\strictwidowpenaltiesthree}
\installaligncommand{strictwidows:4}{\strictwidowpenaltiesfour }

```

We also have four such ‘strict’ definitions for club but only one for broken penalties. For orphan penalties we have four ‘less’ orphan penalties but for widow, club and broken we have only one. So we end up with `lessorphans`, `lessorphans:2`, `lessorphans:3`, `lessorphans:4`, `defaultwidows`, `defaultclubs`, `defaultbroken`, `strictwidows`, `strictwidows:2`, `strictwidows:3`, `strictwidows:4`, `strictclubs`, `strictclubs:2`, `strictclubs:3`, `strictclubs:4` and `strictbroken`.

<p>The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.</p> <p>The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.</p>	<p>The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.</p> <p>The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.</p>
\normalizeparmode 8	\normalizeparmode 8
<p>The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.</p> <p>The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.</p>	<p>The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.</p> <p>The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.</p>
newline and \normalizeparmode 8	newline and \normalizeparmode 8

Figure 11.13 Penalty lists

You can also use `\specificationdef` for other constructs that have this multiple variable setup. Now to `par` passes. This is a mechanism unique to LuaMetaTeX that permits more

todo: copy some from article when published

11.13 Colofon

Author	Hans Hagen
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com contextgarden.net ntg-context@ntg.nl

12 Alignments

low level

TEX

alignments

Contents

12.1	Introduction	204
12.2	Between the lines	206
12.3	Pre-, inter- and post-tab skips	208
12.4	Cell widths	211
12.5	Plugins	212
12.6	Pitfalls and tricks	215
12.7	Rows	218
12.8	Templates	220
12.9	Pitfalls	221
12.10	Remark	223

12.1 Introduction

`\halign` has a couple of subsystems and `\alignments` is one of them. This mechanism is used to

```
\halign {
  \alignmark\hss \aligntab
  \hss\alignmark\hss \aligntab
  \hss\alignmark      \cr
  1.1      \aligntab 2,2      \aligntab 3=3      \cr
  11.11    \aligntab 22,22    \aligntab 33=33     \cr
  111.111  \aligntab 222,222 \aligntab 333=333    \cr
}
```

That one doesn't look too complex and comes out as:

```
1.1_      2,2_      3=3_
11.11_    22,22_    33=33_
111.111_  222,222_  333=333_
```

This is how the previous code comes out when we use one of the `\ConTeXt` table mechanism

```
\starttabulate[|l|c|r|]
  \NC 1.1      \NC 2,2      \NC 3=3      \NC \NR
  \NC 11.11    \NC 22,22    \NC 33=33    \NC \NR
  \NC 111.111  \NC 222,222  \NC 333=333  \NC \NR
\stoptabulate
```

```

1.1      2,2      3=3
11.11    22,22    33=33
111.111  222,222  333=333

```

That one looks a bit different with respect to spaces, so let's go back to the low_level variant:

```

\halign {
    \alignmark\hss \aligntab
    \hss\alignmark\hss \aligntab
    \hss\alignmark      \cr
1.1\aligntab      2,2\aligntab      3=3\cr
11.11\aligntab    22,22\aligntab    33=33\cr
111.111\aligntab  222,222\aligntab  333=333\cr
}

```

Here we don't have spaces in the content part and therefore also no spaces in the result:

```

1.1      2,2      3=3
11.11    22,22    33=33
111.111222,222333=333

```

You can automate dealing with unwanted spacing:

```

\halign {
    \ignorespaces\alignmark\unskip\hss \aligntab
    \hss\ignorespaces\alignmark\unskip\hss \aligntab
    \hss\ignorespaces\alignmark\unskip      \cr
1.1      \aligntab 2,2      \aligntab 3=3      \cr
11.11    \aligntab 22,22    \aligntab 33=33    \cr
111.111  \aligntab 222,222 \aligntab 333=333 \cr
}

```

We get:

```

1.1_      2,2_      3=3_
11.11_    22,22_    33=33_
111.111_  222,222_  333=333_

```

By moving the space skipping and cleanup to the so called preamble we don't need to deal with it in the content part. We can also deal with inter-column spacing there:

```

\halign {
  \ignorespaces\alignmark\unskip\hss \tabskip 1em \aligntab
  \hss\ignorespaces\alignmark\unskip\hss \tabskip 1em \aligntab
  \hss\ignorespaces\alignmark\unskip \tabskip 0pt \cr
  1.1 \aligntab 2,2 \aligntab 3=3 \cr
  11.11 \aligntab 22,22 \aligntab 33=33 \cr
  111.111 \aligntab 222,222 \aligntab 333=333 \cr
}

```

```

1.1_      2,2_      3=3_
11.11_    22,22_    33=33_
111.111_  222,222_  333=333_

```

If for the moment we forget about spanning columns (`\span`) and locally ignoring preamble entries (`\omit`) these basic commands are not that complex to deal with. Here we use `\alignmark` but that is just a primitive that we use instead of `#` while `\aligntab` is the same as `&`, but using the characters instead also assumes that they have the cat-code that relates to a parameter and alignment tab (and in `ConTeXt` that is not the case). `T` have-book.

12.2 Between the lines

The individual rows of a horizontal alignment are treated as lines. This means that, as we see in the previous section, the interline spacing is okay. However, that also means that when we mix the lines with rules, the normal `TEX` habits kick in. Take this:

```

\halign {
  \ignorespaces\alignmark\unskip\hss \tabskip 1em \aligntab
  \hss\ignorespaces\alignmark\unskip\hss \tabskip 1em \aligntab
  \hss\ignorespaces\alignmark\unskip \tabskip 0pt \cr
  \noalign{\hrule}
  1.1 \aligntab 2,2 \aligntab 3=3 \cr
  \noalign{\hrule}
  11.11 \aligntab 22,22 \aligntab 33=33 \cr
  \noalign{\hrule}
  111.111 \aligntab 222,222 \aligntab 333=333 \cr
  \noalign{\hrule}
}

```

The result doesn't look pretty and actually, when you see documents produced by `TEX` using alignments you should not be surprised to notice rather ugly spacing. The user (`O`

1.1	2,2	3=3
11.11	22,22	33=33
111.111	222,222	333=333

The_solution_is_often_easy:_

```
\halign {
    \ignorespaces\strut\alignmark\unskip\hss \tabskip 1em \aligntab
    \hss\ignorespaces\strut\alignmark\unskip\hss \tabskip 1em \aligntab
    \hss\ignorespaces\strut\alignmark\unskip \tabskip 0pt \cr
    \noalign{\hrule}
    1.1 \aligntab 2,2 \aligntab 3=3 \cr
    \noalign{\hrule}
    11.11 \aligntab 22,22 \aligntab 33=33 \cr
    \noalign{\hrule}
    111.111 \aligntab 222,222 \aligntab 333=333 \cr
    \noalign{\hrule}
}
```

1.1_	2,2_	3=3_
11.11_	22,22_	33=33_
111.111_	222,222_	333=333_

The_user_will_not_notice_it_but_alignments_put_some_pressure_on_the_general_T_EX_scanner._A

So_let's_summarize_what_happens:_

1. scan_the_preamble_that_defines_the_cells_(where_the_last_one_is_repeated_when_needed)_
2. check_for_\cr,\noalign_or_a_right_brace;_when_a_row_is_entered_scan_for_cells_in_parallel_the_preamble_so_that_cell_specifications_can_be_applied_(then_start_again)_
3. package_the_preamble_based_on_information_with_regards_to_the_cells_in_a_column_
4. apply_the_preamble_packaging_information_to_the_columns_and_also_deal_with_pending_cell_spans_
5. flush_the_result_to_the_current_list,_unless_packages_in_a_box_a_\halign_is_seen_as_paragraph_and_rows_as_lines_(such_a_table_can_split)_

The_second_(repeated)_step_is_complicated_by_the_fact_that_the_scanner_has_to_look_ahead_for_a_\noalign,\cr,\omit_or_\span_and_when_doing_that_it_has_to_expand_what_comes. This_can_give_side_effects_and_often_results_in_obscure_error_messages. When_for_instance_an_\if_is_seen_and_expanded,_the_wrong_branch_can_be_entered. And_when_you_use_protected_macros_embedded_alignment_commands_are_not_seen_at_all;_of_course_they_still_need_to_produce_valid_operations_in_the_current_context._

All these side effects are to be handled in a macro package when it wraps alignments in a high level interface and ConT_EXt does that for you. But because the code doesn't always MetaT_EX the alignment mechanism has been extended a bit over time.

Nesting `\noalign` is normally not permitted (but one can redefine this primitive such that a macro package nevertheless handles it). The first extension permits nested usage of `\noalign`. This has resulted of a little reorganization of the code. A next extension showed up when overload protection was introduced and extra prefixes were added. We can signal the scanner that a macro is actually a `\noalign` variant: ¹⁹

```
\noaligned\protected\def\InBetween{\noalign{...}}
```

Here the `\InBetween` macro will get the same treatment as `\noalign` and it will not trigger an error. This extension resulted in a second bit of reorganization (think of internal command codes and such) but still the original processing of alignments was there.

A third overhaul of the code actually did lead to some adaptations in the way alignments are constructed so let's move on to that.

12.3 Pre-, inter- and post-tab skips

The basic structure of a preamble and row is actually not that complex: it is a mix of tab skip glue and cells (that are just boxes):

```
\tabskip 10pt
\halign {
  \strut\alignmark\tabskip 12pt\aligntab
  \strut\alignmark\tabskip 14pt\aligntab
  \strut\alignmark\tabskip 16pt\cr
  \noalign{\hrule}
  cell 1.1\aligntab cell 1.2\aligntab cell 1.3\cr
  \noalign{\hrule}
  cell 2.1\aligntab cell 2.2\aligntab cell 2.3\cr
  \noalign{\hrule}
}
```

The tab skips are set in advance and apply to the next cell (or after the last one).

¹⁹ One can argue for using the name `\peekaligned` because in the meantime other alignment primitives also can use this property.

TB:10.000	cell_1.1	TB:12.000	cell_1.2	TB:14.000	cell_1.3	TB:16.000
-----------	----------	-----------	----------	-----------	----------	-----------

TB:10.000	cell_2.1	TB:12.000	cell_2.2	TB:14.000	cell_2.3	TB:16.000
-----------	----------	-----------	----------	-----------	----------	-----------

In the ConT_EXt table mechanisms the value of `\tabskip` is zero in most cases. As in:

```
\tabskip 0pt
\halign {
  \strut\alignmark\aligntab
  \strut\alignmark\aligntab
  \strut\alignmark\cr
  \noalign{\hrule}
  cell_1.1\aligntab cell_1.2\aligntab cell_1.3\cr
  \noalign{\hrule}
  cell_2.1\aligntab cell_2.2\aligntab cell_2.3\cr
  \noalign{\hrule}
}
```

When these skips are zero, they still show up in the end:

cell_1.1	cell_1.2	cell_1.3
----------	----------	----------

cell_2.1	cell_2.2	cell_2.3
----------	----------	----------

Normally, in order to achieve certain effects there will be more align entries in the preamble than cells in the table, for instance because you want vertical lines between cells. When these are not used, you can get quite a bit of empty boxes and zero skips. Now course this is seldom a problem, but when you have a test document where you want to show font properties in a table and that font supports a script with some ten thousand glyphs, you can imagine that it accumulates and in LuaT_EX (and LuaMetaT_EX) nodes and T_EXt we get messages on the console that node memory is bumped.²⁰

²⁰ I suppose it was a coincidence that a few weeks after these features came available a user consulted the mailing list about a few thousand page table that made the engine run out of memory, something that could be cured by enabling these new features.

After playing a bit with stripping zero tab skips I found that the code would not really benefit from such a feature: lots of extra tests made it quite ugly. As a result a first alternative was to just strip zero skips before an alignment got flushed. At least we're then a bit leaner in the processes that come after it. This feature is now available as one of the normalizer bits.

But, as we moved on, a more natural approach was to keep the skips in the preamble, because that is where a guaranteed alternating skip/box is assumed. It also makes that the original documentation is still valid. However, in the rows construction we can be lean. This is driven by a keyword to `\halign`:

```
\tabskip 0pt
\halign noskips {
  \strut\alignmark\aligntab
  \strut\alignmark\aligntab
  \strut\alignmark\cr
  \noalign{\hrule}
  cell 1.1\aligntab cell 1.2\aligntab cell 1.3\cr
  \noalign{\hrule}
  cell 2.1\aligntab cell 2.2\aligntab cell 2.3\cr
  \noalign{\hrule}
}
```

No zero tab skips show up here:

cell_1.1cell_1.2cell_1.3

cell_2.1cell_2.2cell_2.3

When playing with all this the LuaMetaTeX engine also got a tracing option for alignments. = 2 results in:

```
<preamble>
\glue[ignored][...] 0.0pt
\alignrecord
..{\strut }
..<content>
```

```

..\endtemplate }
\glue[ignored][...] 0.0pt
\alignrecord
..\strut }
..<content>
..\endtemplate }
\glue[ignored][...] 0.0pt
\alignrecord
..\strut }
..<content>
..\endtemplate }
\glue[ignored][...] 0.0pt

```

The `ignored_subtype_is` (currently) only used for these alignment tab skips and it triggers a check later on when the rows are constructed. The `<content>` is what get injected in the cell (represented by `\alignmark`). The pseudo primitives are internal and not public.

12.4 Cell widths

Imagine this:

```

\halign {
  x\hbox to 3cm{\strut \alignmark\hss}\aligntab
  x\hbox to 3cm{\strut\hss\alignmark\hss}\aligntab
  x\hbox to 3cm{\strut\hss\alignmark }\cr
  cell 1.1\aligntab cell 1.2\aligntab cell 1.3\cr
  cell 2.1\aligntab cell 2.2\aligntab cell 2.3\cr
}

```

which renders as:

xcell_1.1	x	cell_1.2	x	cell_1.3
xcell_2.1	x	cell_2.2	x	cell_2.3

A reason to have boxes here is that it enforces a cell width but that is done at the cost of an extra wrapper. In LuaMetaTeX the `hlist` nodes are rather large because we have

```

\halign {
  x\tabsize 3cm\strut \alignmark\hss\aligntab

```

```

x\tabsize 3cm\strut\hss\alignmark\aligntab
x\tabsize 3cm\strut\hss\alignmark\hss\cr
cell 1.1\aligntab cell 1.2\aligntab cell 1.3\cr
cell 2.1\aligntab cell 2.2\aligntab cell 2.3\cr
}

```

If you look carefully you will see that this time we don't have the embedded boxes:

xcell_1.1	x cell_1.2	x cell_1.3
xcell_2.1	x cell_2.2	x cell_2.3

So, both the sparse skip and new `\tabsize` feature help to make these extreme tables (spanning hundreds of pages) not consume irrelevant memory and also make that later on we don't have to consult useless nodes.

12.5 Plugins

Yet another LuaMetaTeX extension is a callback that kicks in between the preamble preroll and TeXt table mechanisms.

```

\starttabulate[|lG{.}|cG{,}|rG{=}|cG{x}|]
\NC 1.1      \NC 2,2      \NC 3=3      \NC a 0xFF    \NC \NR
\NC 11.11    \NC 22,22    \NC 33=33    \NC b 0xFFF    \NC \NR
\NC 111.111  \NC 222,222  \NC 333=333  \NC c 0xFFFF  \NC \NR
\stoptabulate

```

The tabulate mechanism in ConTeXt is rather old and stable and it is the preferred way to

```

1.1      2,2      3=3      a 0xFF
11.11    22,22    33=33    b 0xFFF
111.111  222,222  333=333  c 0xFFFF

```

Let's make clear that this is *not* an engine feature but a ConTeXt one. It is however made e

```

\halign noskips \alignmentcharactertrigger \bgroup
\tabskip2em
\setalignmentcharacter.\ignorespaces\alignmark\unskip\hss \aligntab
\hss\setalignmentcharacter.\ignorespaces\alignmark\unskip\hss \aligntab
\hss\setalignmentcharacter=\ignorespaces\alignmark\unskip \aligntab
\hss \ignorespaces\alignmark\unskip\hss \cr
1.1 \aligntab 2,2 \aligntab 3=3 \aligntab \setalignmentcharacter{.}\relax 4.4\cr
11.11 \aligntab 22,22 \aligntab 33=33 \aligntab \setalignmentcharacter{,}\relax 44,44\cr
111.111 \aligntab 222,222 \aligntab 333=333 \aligntab \setalignmentcharacter{!}\relax 444!444\cr

```

```

x \aligntab      x \aligntab      x \aligntab \setalignmentcharacter{/}\relax /\cr
.1 \aligntab     ,2 \aligntab     =3 \aligntab \setalignmentcharacter{?}\relax ?4\cr
.111 \aligntab   ,222 \aligntab   =333 \aligntab \setalignmentcharacter{=}\relax 44=444\cr
\egroup

```

This_rather_verbose_setup_renders_as:_

1.1_	2,2_	3=3_	4.4
11.11_	22,22_	33=33_	44,44
111.111_	222,222_	333=333_	444!444
x_	x_	x_	/
.1_	,2_	=3_	?4
.111_	,222_	=333_	44=444

Using_a_high_level_interface_makes_sense_but_local_control_over_such_alignment_too,_so_here_follow_some_more_examples._Here_we_use_different_alignment_characters:_

```

\starttabulate[|lG{.}|cG{,}|rG{=}|cG{x}|]
\NC 1.1      \NC 2,2      \NC 3=3      \NC a 0xFF      \NC \NR
\NC 11.11    \NC 22,22    \NC 33=33    \NC b 0xFFF      \NC \NR
\NC 111.111  \NC 222,222  \NC 333=333  \NC c 0xFFFF    \NC \NR
\stoptabulate

```

1.1	2,2	3=3	a 0xFF
11.11	22,22	33=33	b 0xFFF
111.111	222,222	333=333	c 0xFFFF

In_this_example_we_specify_the_characters_in_the_cells._We_still_need_to_add_a_specifier_in_the_preamble_definition_because_that_will_trigger_the_plugin._

```

\starttabulate[|lG{}|rG{}|]
\NC left      \NC right      \NC \NR
\NC \showglyphs \setalignmentcharacter{.}1.1 \NC \setalignmentcharacter{.}1.1 \NC \NR
\NC \showglyphs \setalignmentcharacter{,}11,11 \NC \setalignmentcharacter{,}11,11 \NC \NR
\NC \showglyphs \setalignmentcharacter{=}111=111 \NC \setalignmentcharacter{=}111=111 \NC \NR
\stoptabulate

```

left	right
1.1	1.1
11,11	11,11
111=111	111=111

You_can_mix_these_approaches:_

```

\starttabulate[|lG{.}|rG{}|]
\NC left      \NC right      \NC \NR

```

```

\NC 1.1      \NC \setalignmentcharacter{.}1.1      \NC\NR
\NC 11.11    \NC \setalignmentcharacter{.}11.11    \NC\NR
\NC 111.111  \NC \setalignmentcharacter{.}111.111  \NC\NR
\stoptabulate

```

```

left      right
  1.1      1.1
 11.11    11.11
111.111  111.111

```

Here the already present alignment feature, that at some point in `tabulate` might use this new feature, is meant for numbers, but here we can go wild with words, although of course you need to keep in mind that we deal with typeset text, so there may be no match.

```

\starttabulate[|lG{.}|rG{.}|]
\NC foo.bar \NC foo.bar \NC \NR
\NC oo.ba   \NC oo.ba   \NC \NR
\NC o.b     \NC o.b     \NC \NR
\stoptabulate

```

```

foo.bar  foo.bar
oo.ba    oo.ba
o.b      o.b

```

This feature will only be used in know situations and those seldom involve advanced typesetting. However, the following does work: ²¹

```

\starttabulate[|cG{d}|]
\NC \smallcaps abcdefgh \NC \NR
\NC          xdy        \NC \NR
\NC \sl          xdy     \NC \NR
\NC \tttf        xdy     \NC \NR
\NC \tfd         d       \NC \NR
\stoptabulate

```

```

abc d efgh
  x d y
  x d y

```

²¹ Should this be an option instead?

x d y
d

As always with such mechanisms, the question is “Where to stop?” But it makes for nice demos and as long as little code is needed it doesn't hurt.

12.6 Pitfalls and tricks

The next example mixes bidirectional typesetting. It might look weird at first sight but the result conforms to what we discussed in previous paragraphs.

```
\starttabulate[|lG{.}|lG{}|]
\NC \righttoleft 1.1 \NC \righttoleft \setalignmentcharacter{.}1.1 \NC\NR
\NC 1.1 \NC \setalignmentcharacter{.}1.1 \NC\NR
\NC \righttoleft 1.11 \NC \righttoleft \setalignmentcharacter{.}1.11 \NC\NR
\NC 1.11 \NC \setalignmentcharacter{.}1.11 \NC\NR
\NC \righttoleft 1.111 \NC \righttoleft \setalignmentcharacter{.}1.111 \NC\NR
\NC 1.111 \NC \setalignmentcharacter{.}1.111 \NC\NR
\stoptabulate
```

```
1.1 1.1
1.1 1.1
11.1 11.1
1.11 1.11
111.1 111.1
1.111 1.111
```

In case of doubt, look at this:

```
\starttabulate[|lG{.}|lG{}|lG{.}|lG{}|]
\NC \righttoleft 1.1 \NC \righttoleft \setalignmentcharacter{.}1.1 \NC
1.1 \NC \setalignmentcharacter{.}1.1 \NC\NR
\NC \righttoleft 1.11 \NC \righttoleft \setalignmentcharacter{.}1.11 \NC
1.11 \NC \setalignmentcharacter{.}1.11 \NC\NR
\NC \righttoleft 1.111 \NC \righttoleft \setalignmentcharacter{.}1.111 \NC
1.111 \NC \setalignmentcharacter{.}1.111 \NC\NR
\stoptabulate
```

```
1.1 1.1 1.1 1.1
11.1 11.1 1.11 1.11
111.1 111.1 1.111 1.111
```

The next example shows the effect of `\omit` and `\span`. The first one makes that in this cell the preamble template is ignored.

`\halign \bgroup`

```

\tabsize 2cm\relax      [\alignmark]\hss \aligntab
\tabsize 2cm\relax \hss[\alignmark]\hss \aligntab
\tabsize 2cm\relax \hss[\alignmark]\cr
      1\aligntab        2\aligntab        3\cr
\omit 1\aligntab \omit 2\aligntab \omit 3\cr
      1\aligntab        2\span            3\cr
      1\span            2\aligntab        3\cr
      1\span            2\span            3\cr
      1\span \omit      2\span \omit      3\cr
\omit 1\span \omit      2\span \omit      3\cr

```

`\egroup`

Spans are applied at the end so you see a mix of templates applied.

Handwriting practice sheet for the number 123. The sheet contains six rows of handwriting lines (top solid, middle dashed, bottom solid). Each row shows a different arrangement of the numbers 1, 2, and 3, some enclosed in boxes and some with arrows indicating stroke direction. The numbers are written in a simple, bold, sans-serif font.

Row	Content
1	[1] [2] [3]
2	1 2 3
3	[1] [2] [3]
4	[1] [2] [3]
5	[1] [2] [3]
6	123

When you define an alignment inside a macro, you need to duplicate the `\alignmark` signals. This is similar to embedded macro definitions. But in `LuaMetaTeX` we can get around this

\halign \bgroup

```
\tabsize 2cm\relax \aligncontent\hss \alignstab
\tabsize 2cm\relax \hss\aligncontent\hss \alignstab
\tabsize 2cm\relax \hss\aligncontent\cr
1\alignstab 2\alignstab 3\cr
```

```
A\aligntab B\aligntab C\cr
\egroup
```

```
1          2          3
A          B          C
```

In this example we still have to be verbose in the way we align but we can do this:

```
\halign \bgroup
\tabsize 2cm\relax \aligncontentleft \aligntab
\tabsize 2cm\relax \aligncontentmiddle\aligntab
\tabsize 2cm\relax \aligncontentright \cr
1\aligntab 2\aligntab 3\cr
A\aligntab B\aligntab C\cr
\egroup
```

Where the helpers are defined as:

```
\noaligned\protected\def\aligncontentleft
{\ignorespaces\aligncontent\unskip\hss}

\noaligned\protected\def\aligncontentmiddle
{\hss\ignorespaces\aligncontent\unskip\hss}

\noaligned\protected\def\aligncontentright
{\hss\ignorespaces\aligncontent\unskip}
```

The preamble scanner see such macros as candidates for a single level expansion so it will inject the meaning and see the `\aligncontent` eventually.

```
1          2          3
A          B          C
```

The same effect could be achieved by using the `\span` prefix:

```
\def\aligncontentleft{\ignorespaces\aligncontent\unskip\hss}

\halign { ... \span\aligncontentleft ... }
```

One of the reasons for not directly using the low level `\halign` command is that it's a lot of work but by providing a set of helpers like here might change that a bit. Keep in mind that much of the above is not new in the sense that we could not achieve the same already, it's just a bit programmer friendly.

12.7 Rows

Alignment support is what the documented source calls ‘interwoven’. When the engine scans for input it processing text, math or alignment content. While doing alignments it collects rows, and inside these cells but also deals with material that ends up in between. In LuaMetaTeX I tried to isolate the bits and pieces as good as possible but set’ state.

Scanning starts with interpreting the preamble, and then grabbing rows. There is some nasty lookahead involved for `\noalign`, `\span`, `\omit`, `\cr` and `\crrc` and that is not code one wants to tweak too much (although we did in LuaMetaTeX). This means for ‘start a row here’ primitive is sort of tricky (but it might happen some day) which in turn means that it is not really possible to set row properties. As an experiment we can set some properties now by hijacking `\noalign` and storing them on the alignment stack (indeed: at the cost of some extra overhead and memory). This permits the following:

```
\halign {
  \hss
  \ignorespaces \alignmark \removeunwantedspaces
  \hss
  \quad \aligntab \quad
  \hss
  \ignorespaces \alignmark \removeunwantedspaces
  \hss
  \cr
  \noalign xoffset 40pt {}
  {\darkred cell one} \aligntab {\darkgray cell one} \cr
  \noalign orientation "002 {}
  {\darkgreen cell one} \aligntab {\darkblue cell one} \cr
  \noalign xoffset 40pt {}
  {\darkred cell two} \aligntab {\darkgray cell two} \cr
  \noalign orientation "002 {}
  {\darkgreen cell two} \aligntab {\darkblue cell two} \cr
  \noalign xoffset 40pt {}
  {\darkred cell three} \aligntab {\darkgray cell three} \cr
  \noalign orientation "002 {}
  {\darkgreen cell three} \aligntab {\darkblue cell three} \cr
  \noalign xoffset 40pt {}
  {\darkred cell four} \aligntab {\darkgray cell four} \cr
  \noalign orientation "002 {}
```

```

{\darkgreen cell four} \aligntab {\darkblue cell four} \cr
}

```

cell one	cell one
cell one	cell one
cell two	cell two
cell two	cell two
cell three	cell three
cell three	cell three
cell four	cell four
cell four	cell four

The supported keywords are similar to those for boxes: source, target, anchor, orientation, dimensions can be prefixed by add and reset wipes all. Here is another example:

```

\halign {
  \hss
  \ignorespaces \alignmark \removeunwantedspaces
  \hss
  \quad \aligntab \quad
  \hss
  \ignorespaces \alignmark \removeunwantedspaces
  \hss
  \cr
  \noalign xmove 40pt {}
  {\darkred cell one} \aligntab {\darkgray cell one} \cr
  {\darkgreen cell one} \aligntab {\darkblue cell one} \cr
  \noalign xmove 20pt {}
  {\darkred cell two} \aligntab {\darkgray cell two} \cr
  {\darkgreen cell two} \aligntab {\darkblue cell two} \cr
  \noalign xmove 40pt {}
  {\darkred cell three} \aligntab {\darkgray cell three} \cr
  {\darkgreen cell three} \aligntab {\darkblue cell three} \cr
  \noalign xmove 20pt {}
  {\darkred cell four} \aligntab {\darkgray cell four} \cr
  {\darkgreen cell four} \aligntab {\darkblue cell four} \cr
}

```

cell one	cell one
cell one	cell one
cell two	cell two
cell two	cell two
cell three	cell three
cell three	cell three
cell four	cell four
cell four	cell four

Some more features might be added in the future as is it an interesting playground. It is to be seen how this ends up in ConT_EXt high level interfaces like tabulate.

12.8 Templates

The `\omit` command signals that the template should not be applied. But what if we actually want something at the left and right of the content? Here is how it's done:

```
\tabskip10pt \showboxes
\halign\bgroup
  [\hss\aligncontent\hss]\aligntab
  [\hss\aligncontent\hss]\aligntab
  [\hss\aligncontent\hss]\cr
      x\aligntab                x\aligntab        x\cr
      xx\aligntab               xx\aligntab       xx\cr
      xxx\aligntab              xxx\aligntab      xxx\cr
\omit   oo\aligntab\omit        oo\aligntab\omit  oo\cr
      xx\aligntab\realign{\hss({})\hss}xx\aligntab  xx\cr
\realign{\hss({})\hss}xx\aligntab xx\aligntab xx\cr
\egroup
```

The `\realign` command is like an `omit` but it expects two token lists that will for this cell be used instead of the ones from the preamble. While `\omit` also skips insertion of `\everytab`, here it is inserted, just like with normal preambles.

X	X	X
XX	XX	XX
XXX	XXX	XXX
oo	oo	oo
XX	(XX)	XX
(XX)	XX	XX

It will probably take a while before I'll apply this in ConT_EXt because changing existing (sta

12.9 Pitfalls

Alignment have a few properties that can catch you off-guard. One is the use of `\everycr`. The next example demonstrates that it is also injected after the preamble definition.

```
\everycr{\noalign{\hrule}}
\halign\bgroup \hsize 5cm \strut \alignmark\cr one\cr two\cr\egroup
```

This makes sense because it is one way to make sure that for instance a rule gets the width of the cell.

one
two

The same is of course true for a vertical align:

```
\everycr{\noalign{\vrule}}
\valign\bgroup \hsize 4cm \strut \aligncontent\cr one\cr two\cr\egroup
```

We set the width because otherwise the current text width is used.

one	two	
-----	-----	--

Something similar happens with a `\tabskip`: the value set before the alignment is used left of the first cell.

```
\tabskip10pt
\halign\bgroup \tabskip20pt\relax\aligncontent\cr x\cr x\cr \egroup
```

X	
X	

The `\tabskip` outside the alignment is an internal glue register so you can for instance use the prefix `\global`. However, in a preamble it is more a directive: the given value is stored with the cell. This means that the next code is invalid:

```
\tabskip10pt
\halign\bgroup \global\tabskip20pt\relax\aligncontent\cr x\cr x\cr \egroup
```

The parser looks at tokens in the preamble, sees the `\global` and appends it to the current pre-part of the cell's template. Then it sees a `\tabskip` and assigns the value after it to the cell's skip. The token and its value just disappear, they are not appended to the template. Now, when the template is injected (and interpreted) this `\global` expects a variable next and in our case the `x` doesn't qualify. The next snippet however works okay:

```

\scratchcounter0
\halign\bgroup
  \global\tabskip40pt\relax\advance\scratchcounter\plusone\aligncontent
  \cr
  x:\the\scratchcounter\cr
  x:\the\scratchcounter\cr
  x:\the\scratchcounter\cr
\egroup

```

Here_the_\global_is_applied_to_the_advance_because_the_skip_definition_is_not_in_the_preamb

```

x:1
x:2
x:3

```

Here_is_a_variant:_

```

\scratchcounter0
\halign\bgroup
  \global\tabskip10pt\relax\aligncontent\cr
  \advance\scratchcounter\plusone x:\the\scratchcounter\cr
  \advance\scratchcounter\plusone x:\the\scratchcounter\cr
  \advance\scratchcounter\plusone x:\the\scratchcounter\cr
\egroup

```

Again_the_\global_stays_and_this_time_if_ends_up_before_the_content_which_starts_with_an_advance._

```

x:1
x:2
x:3

```

Normally_you_will_not_need_the_next_trickery_but_it_shows_that_cells_are_grouped:_

```

\halign\bgroup\aligncontent\cr
  1\atendofgrouped{A}\atendofgrouped{B}\cr
  2\aftergrouped {A}\aftergrouped {B}\cr
  3 \cr
\egroup

```

Maybe_at_some_point_I'll_add_something_a_bit_more_tuned_for_dealing_with_cells,_but_here_is_what_you_get_with_the_above:_

```

1AB
2

```

AB3_

12.10 Remark

It can be that the way alignments are interfaced with respect to attributes is a bit different between LuaT_EX and LuaMetaT_EX but because the former is frozen (in order not to T_EXt LMTX.

In principle we can have hooks into the rows for pre and post material but it doesn't really pay off as grouping will still interfere. So for now I decided not to add these.

12.10 Colofon

Author	Hans Hagen
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com contextgarden.net ntg-context@ntg.nl

13 Marks

low level

TEX

marks

Contents

13.1	Introduction	225
13.2	The basics	225
13.3	Migration	226
13.4	Tracing	229
13.5	High level commands	230
13.6	Pitfalls	232

13.1 Introduction

Marks are one of the subsystems of \TeX , as are for instance alignments and math as well as

In \MkII marks are used to keep track of colors, transparencies and more properties that work across page boundaries. It permits picking up at the top of a page from where one left at the bottom of the preceding one. When \MkII was written there was only one mark so on top of that a multiple mark mechanism was implemented that filtered specific marks from a collection. Later, $\varepsilon\text{-}\text{\TeX}$ provided mark classes so that me

But, marks have some nasty limitations, so from the \ConTeXt perspective there always was. \TeX engine has a `\clearmarks` primitive but that works global. In \LuaMetaTeX a proper mar

In \MkIV the engine's marks were not used at all and an alternative mechanism was written using Lua. It actually is one of the older \MkIV features. It doesn't have the side effects that native marks have but it comes at the price of more overhead, although that is bearable.

In this document we discuss marks but assume that \LuaMetaTeX is used with \ConTeXt LMTX. There we experiment with using the native marks, complemented by a few Lua

13.2 The basics

Although the original \TeX primitives are there, the plural $\varepsilon\text{-}\text{\TeX}$ mark commands are to be u

```
\marks0{This is mark 0} % equivalent to: \mark{This is mark 0}
\marks4{This is mark 4}
```

²² This is probably true for most \LuaTeX and \LuaMetaTeX extensions, maybe example usage create retrospective demand. But one reason for picking up on engine development is that in the \ConTeXt perspective we actually had some demands.

When a page has been split off, you can (normally this only makes sense in the output routine) access marks with:

```
\topmarks 4
\firstmarks4
\botmarks 4
```

A ‘top’ mark is the last one on the previous page(s), the ‘first’ and ‘bottom’ refer to the current page. A mark is a so called node, something that ends up in the current list and the token list is stored with it. The accessors are just commands and they fetch the token list from a separately managed storage. When you set or access a mark that has not yet been used, the storage is bumped to the right size, so it doesn't make sense to use e.g. `\marks 999` when there are no 998 ones too: it not only takes memory, it also makes \TeX run over all these mark stores when synchronization is needed.

```
\newmarks\MyMark
```

Currently the first 16 marks are skipped so this makes `\MyMark` become mark 17. The reason is that we want to make sure that users who experiment with marks have some scratch marks available and don't overload system defined ones. Future versions of \ConTeXt might become more restrictive.

Marks can be cleared with:

```
\clearmarks 4
```

which clears the storage that keeps the top, first and bot marks. This happens immediately. You can delay this by putting a signal in the list:

```
\flushmarks 4
```

This (LuaMeta \TeX) feature makes it for instance easy to reset marks that keep track of sections. \TeX always had that.

The current, latest assigned, value of a mark is available too:

```
\currentmarks 4
```

Using this value in for instance headers and footers makes no sense because the last node set can be on a following page.

13.3 Migration

In the introduction we mentioned that LuaMeta \TeX has migration built in. In MkIV we have

Migrated_marks_end_up_in_the_postmigrated_sublist_of_a_box. In other lowlevel manuals we discuss these pre- and postmigrated_sublists. As example we use this definition:

```
\setbox0\vbox\bgroup
test \marks 4 {mark 4.1}\par
test \marks 4 {mark 4.1}\par
test \marks 4 {mark 4.1}\par
\egroup
```

When we turn migration on (officially the second bit):

```
\automigrationmode"FF \showbox0
```

we get this:

```
> \box0=
2:4: \vbox[normal][...], width 483.69687, height 63.43475, depth 0.15576, direction l2r
2:4: .\list
2:4: ..\hbox[line][...], width 483.69687, height 7.48193, depth 0.15576, glue 459.20468fil, direction l2r
2:4: ...list
2:4: ....\glue[left hang][...] 0.0pt
2:4: ....\glue[left][...] 0.0pt
2:4: ....\glue[parfillleft][...] 0.0pt
2:4: ....\par[newgraf][...], hangafter 1, hsize 483.69687, pretolerance 100, tolerance 200, adjdemerits 10000, linepenalty 10, doublehyphendemerits 10000,
    finalhyphendemerits 5000, clubpenalty 2000, widowpenalty 2000, brokenpenalty 100, parfillskip 0.0pt plus 1.0fil, hyphenationmode 499519
2:4: ....\glue[indent][...] 0.0pt
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000065 e
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000073 s
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ....\glue[space][...] 3.49658pt plus 1.74829pt minus 1.16553pt, font 8
2:4: ....\penalty[line][...] 10000
2:4: ....\glue[parfill][...] 0.0pt plus 1.0fil
2:4: ....\glue[right][...] 0.0pt
2:4: ....\glue[right hang][...] 0.0pt
2:4: ..\glue[par][...] 11.98988pt plus 3.99663pt minus 3.99663pt
2:4: ..\glue[baseline][...] 8.34883pt
2:4: ..\hbox[line][...], width 483.69687, height 7.48193, depth 0.15576, glue 459.20468fil, direction l2r
2:4: ...list
2:4: ....\glue[left hang][...] 0.0pt
2:4: ....\glue[left][...] 0.0pt
2:4: ....\glue[parfillleft][...] 0.0pt
2:4: ....\par[newgraf][...], hangafter 1, hsize 483.69687, pretolerance 100, tolerance 200, adjdemerits 10000, linepenalty 10, doublehyphendemerits 10000,
    finalhyphendemerits 5000, clubpenalty 2000, widowpenalty 2000, brokenpenalty 100, parfillskip 0.0pt plus 1.0fil, hyphenationmode 499519
2:4: ....\glue[indent][...] 0.0pt
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000065 e
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000073 s
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ....\glue[space][...] 3.49658pt plus 1.74829pt minus 1.16553pt, font 8
2:4: ....\penalty[line][...] 10000
2:4: ....\glue[parfill][...] 0.0pt plus 1.0fil
2:4: ....\glue[right][...] 0.0pt
2:4: ....\glue[right hang][...] 0.0pt
2:4: ..\glue[par][...] 11.98988pt plus 3.99663pt minus 3.99663pt
2:4: ..\glue[baseline][...] 8.34883pt
2:4: ..\hbox[line][...], width 483.69687, height 7.48193, depth 0.15576, glue 459.20468fil, direction l2r
2:4: ...list
2:4: ....\glue[left hang][...] 0.0pt
2:4: ....\glue[left][...] 0.0pt
2:4: ....\glue[parfillleft][...] 0.0pt
2:4: ....\par[newgraf][...], hangafter 1, hsize 483.69687, pretolerance 100, tolerance 200, adjdemerits 10000, linepenalty 10, doublehyphendemerits 10000,
    finalhyphendemerits 5000, clubpenalty 2000, widowpenalty 2000, brokenpenalty 100, parfillskip 0.0pt plus 1.0fil, hyphenationmode 499519
2:4: ....\glue[indent][...] 0.0pt
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000065 e
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000073 s
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ....\glue[space][...] 3.49658pt plus 1.74829pt minus 1.16553pt, font 8
2:4: ....\penalty[line][...] 10000
2:4: ....\glue[parfill][...] 0.0pt plus 1.0fil
2:4: ....\glue[right][...] 0.0pt
2:4: ....\glue[right hang][...] 0.0pt
```

```

2:4: .\postmigrated
2:4: ..\mark[4][...]
2:4: ..{mark 4.1}
2:4: ..\mark[4][...]
2:4: ..{mark 4.1}
2:4: ..\mark[4][...]
2:4: ..{mark 4.1}

```

When we don't migrate, enforced with:

\automigrationmode"00 \showbox0

the result is:

```

> \box0=
2:4: \vbox[normal][...], width 483.69687, height 63.43475, depth 0.15576, direction l2r
2:4: .\list
2:4: ..\hbox[line][...], width 483.69687, height 7.48193, depth 0.15576, glue 459.20468fil, direction l2r
2:4: ...list
2:4: ....\glue[left hang][...] 0.0pt
2:4: ....\glue[left][...] 0.0pt
2:4: ....\glue[parfillsleft][...] 0.0pt
2:4: ....\par[newgraf][...], hangafter 1, hsize 483.69687, pretolerance 100, tolerance 200, adjdemerits 10000, linepenalty 10, doublehyphendemerits 10000,
    finalhyphendemerits 5000, clubpenalty 2000, widowpenalty 2000, brokenpenalty 100, parfillskip 0.0pt plus 1.0fil, hyphenationmode 499519
2:4: ....\glue[indent][...] 0.0pt
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000065 e
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000073 s
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ....\glue[space][...] 3.49658pt plus 1.74829pt minus 1.16553pt, font 8
2:4: ....\penalty[line][...] 10000
2:4: ....\glue[parfills][...] 0.0pt plus 1.0fil
2:4: ....\glue[right][...] 0.0pt
2:4: ....\glue[right hang][...] 0.0pt
2:4: ..\mark[4][...]
2:4: ..{mark 4.1}
2:4: ..\glue[par][...] 11.98988pt plus 3.99663pt minus 3.99663pt
2:4: ..\glue[baseline][...] 8.34883pt
2:4: ..\hbox[line][...], width 483.69687, height 7.48193, depth 0.15576, glue 459.20468fil, direction l2r
2:4: ...list
2:4: ....\glue[left hang][...] 0.0pt
2:4: ....\glue[left][...] 0.0pt
2:4: ....\glue[parfillsleft][...] 0.0pt
2:4: ....\par[newgraf][...], hangafter 1, hsize 483.69687, pretolerance 100, tolerance 200, adjdemerits 10000, linepenalty 10, doublehyphendemerits 10000,
    finalhyphendemerits 5000, clubpenalty 2000, widowpenalty 2000, brokenpenalty 100, parfillskip 0.0pt plus 1.0fil, hyphenationmode 499519
2:4: ....\glue[indent][...] 0.0pt
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000065 e
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000073 s
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ....\glue[space][...] 3.49658pt plus 1.74829pt minus 1.16553pt, font 8
2:4: ....\penalty[line][...] 10000
2:4: ....\glue[parfills][...] 0.0pt plus 1.0fil
2:4: ....\glue[right][...] 0.0pt
2:4: ....\glue[right hang][...] 0.0pt
2:4: ..\mark[4][...]
2:4: ..{mark 4.1}
2:4: ..\glue[par][...] 11.98988pt plus 3.99663pt minus 3.99663pt
2:4: ..\glue[baseline][...] 8.34883pt
2:4: ..\hbox[line][...], width 483.69687, height 7.48193, depth 0.15576, glue 459.20468fil, direction l2r
2:4: ...list
2:4: ....\glue[left hang][...] 0.0pt
2:4: ....\glue[left][...] 0.0pt
2:4: ....\glue[parfillsleft][...] 0.0pt
2:4: ....\par[newgraf][...], hangafter 1, hsize 483.69687, pretolerance 100, tolerance 200, adjdemerits 10000, linepenalty 10, doublehyphendemerits 10000,
    finalhyphendemerits 5000, clubpenalty 2000, widowpenalty 2000, brokenpenalty 100, parfillskip 0.0pt plus 1.0fil, hyphenationmode 499519
2:4: ....\glue[indent][...] 0.0pt
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000065 e
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000073 s
2:4: ....\glyph[32768][...], language (n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <8: DejaVuSerif @ 11.0pt>, glyph U+000074 t
2:4: ....\glue[space][...] 3.49658pt plus 1.74829pt minus 1.16553pt, font 8
2:4: ....\penalty[line][...] 10000
2:4: ....\glue[parfills][...] 0.0pt plus 1.0fil
2:4: ....\glue[right][...] 0.0pt
2:4: ....\glue[right hang][...] 0.0pt
2:4: ..\mark[4][...]
2:4: ..{mark 4.1}

```

When you say \showmakeup_or_in_this_case_\showmakeup[mark] the marks are visualized:

test	test
test	test
test	test
enabled	disabled

Here `_sm` means ‘set mark’ while `_rm` would indicate a ‘reset mark’. Of course migrated marks don't show up because these are bound to the box and thereby have become a specific box property as can be seen in the above trace.

13.4 Tracing

The `LuaMetaTeX` engine has a dedicated tracing option for marks. The fact that the tradition

```
\tracingmarks1
\tracingonline2
```

When tracing is set to 1 we get a list of marks for the just split of page:

```
2:7: <mark class 51, top := bot>
2:7: ..{sample 9.1}
2:7: <mark class 51: first := mark>
2:7: ..{sample 10.1}
2:7: <mark class 51: bot := mark>
2:7: ..{sample 10.1}
2:7: <mark class 51, page state>
2:7: ..top {sample 9.1}
2:7: ..first {sample 10.1}
2:7: ..bot {sample 10.1}
```

When tracing is set to 2 you also get details we get a list of marks of the analysis:

```
1:9: <mark class 51, top := bot>
1:9: ..{sample 5.1}
1:9: <mark class 51: first := mark>
1:9: ..{sample 6.1}
1:9: <mark class 51: bot := mark>
1:9: ..{sample 6.1}
1:9: <mark class 51: bot := mark>
1:9: ..{sample 7.1}
1:9: <mark class 51: bot := mark>
1:9: ..{sample 8.1}
```

```

1:9: <mark class 51: bot := mark>
1:9: ..{sample 9.1}
1:9: <mark class 51, page state>
1:9: ..top {sample 5.1}
1:9: ..first {sample 6.1}
1:9: ..bot {sample 9.1}

```

13.5 High level commands

I think that not that many users define their own marks. They are useful for showing section related titles in headers and footers but the implementation of that is hidden. The native mark references are top, first and bottom but in the ConT_EXt interface

ConT _E Xt	T _E X	column	page
previous	top	last before sync	last on previous page
top	first	first in sync	first on page
bottom	bot	last in sync	last on page
first	top	first not top in sync	first on page
last	bot	last not bottom in sync	last on page
default		the same as first	
current		the last set value	

In order to separate marks in ConT_EXt from those in T_EX, the term ‘marking’ is used. In MkIV the regular marks mechanism is of course there but, as mentioned, not used. By using a different namespace we could make the transition from MkII to MkIV (the same is true for some more mechanisms).

A marking is defined with

```
\definemarking[MyMark]
```

A defined marking can be set with two equivalent commands:

```

\setmarking[MyMark]{content}
\marking    [MyMark]{content}

```

The content is not typeset but stored as token list. In the sectioning mechanism that uses markings we don't even store titles, we store a reference to a title. In order to use that (deep down) we hook in a filter command. By default that command does nothing:

```
\setupmarking[MyMark][filtercommand=\firstofoneargument]
```

The_token_list_does_not_get_expanded_by_default,_unless_you_set_it_up:_

```
\setupmarking[MyMark][expansion=yes]
```

The_current_state_of_a_marking_can_be_cleared_with:_

```
\clearmarking[MyMark]
```

but_because_that_en_is_not_synchronized_the_real_deal_is:_

```
\resetmarking[MyMark]
```

Be_aware_that_it_introduces_a_node_in_the_list._You_can_test_if_a_marking_is_defined_with_(as_usual)_a_test_macro._Contrary_to_(most)_other_test_macros_this_one_is_fully_expandable:_

```
\doifelsemarking {MyMark} {
    defined
} {
    undefined
}
```

Because_there_can_be_a_chain_involved,_we_can_relate_markings._Think_of_sections_below_chapters_and_subsections_below_sections:_

```
\relatemarking[MyMark][YourMark]
```

When_a_marking_is_set_its_relatives_are_also_reset,_so_setting_YourMark_will_reset_MyMark._It_is_this_kind_of_features_that_made_for_marks_being_wrapped_into_high_level_commands_very_early_in_the_ConT_EXt_development_(and_one_can_even_argue_that_this_is_why_a_package_T_EXt_exists_in_the_first_place)._

The_rest_of_the_(relatively_small)_repertoire_of_commands_has_to_do_with_fetching_markings._The_general_command_is_\getmarking_that_takes_two_or_three_arguments:_

```
\getmarking[MyMarking][first]
\getmarking[MyMarking][page][first]
\getmarking[MyMarking][page][first]
\getmarking[MyMarking][column:1][first]
```

There_are_(normally)_three_marks_that_can_be_fetched_so_we_have_three_commands_that_do_just_that:_

```
\fetchonemark [MyMarking][which one]
\fetchtwomarks [MyMarking]
\fetchallmarks [MyMarking]
```

You can setup a separator key which by default is:

```
\setupmarking [MyMarking][separator=\space\emdash\space]
```

Injection is enabled by default due to this default:

```
\setupmarking [MyMarking][state=start]
```

The following three variants are (what is called) fully expandable:

```
\fetchonemarking [MyMarking][which one]
\fetchtwomarkings [MyMarking]
\fetchallmarkings [MyMarking]
```

13.6 Pitfalls

The main pitfall is that a (re)setting a mark will inject a node which in vertical mode can interfere with spacing. In for instance section commands we wrap them with the title so there it should work out okay.

13.6 Colofon

Author	Hans Hagen
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com contextgarden.net ntg-context@ntg.nl

14 Inserts

low level

TEX

inserts

Contents

14.1	Introduction	234
14.2	The page builder	234
14.3	Inserts	236
14.4	Storing	237
14.5	Synchronizing	237
14.6	Migration	237
14.7	Callbacks	237

14.1 Introduction

This document is a mixed bag. We do discuss inserts but also touch elements of the page builder because inserts and regular page content are handled there. Examples of mechanisms that use inserts are footnotes. These have an anchor in the running text and some content that ends up (normally) at the bottom of the page. When considering a page break the engine tries to make sure that the anchor (reference) and the content end up on the same page. When there is too much, it will distribute (split) the content over pages.

We can discuss page breaks in a (pseudo) scientific way and explore how to optimize this process, taking into accounts also inserts that contain images but it doesn't make much sense to do that because in practice we can encounter all kind of interferences. Theory and practice are too different because a document can contain a wild mix of text, figures, formulas, notes, have backgrounds and location dependent processing. It gets even more complex when we are dealing with columns because \TeX doesn't really know the

I will therefore stick to some practical aspects and the main reason for this document is that I sort of document engine features and at the same time give an impression of what we deal with. I will do that in the perspective of \LuaMetaTeX , which has a few more

Currently this document is mostly for myself to keep track of the state of inserts and the page builder in \LuaMetaTeX and \ConTeXt LMTX . The text is not yet corrected and can be

14.2 The page builder

When your document is processed content eventually gets added to the so called main vertical list (mvl). Content first get appended to the list of contributions and at specific moments it will be handed over to the mvl. This process is called page building. There we can encounter the following elements (nodes):

glue	a vertical skip
penalty	a vertical penalty
kern	a vertical kern
vlist	a a vertical box
hlist	a horizontal box (often a line)
rule	a horizontal rule
boundary	a boundary node
whatsit	a node that is used by user code (often some extension)
mark	a token list (as used for running headers)
insert	a node list (as used for notes)

The engine itself will not insert anything other than this but Lua code can mess up the contribution list and the mvl and that can trigger an error. Handing over the contributions is done by the page builder and that one kicks in in several places:

- When a penalty gets inserted it is part of evaluating if the output routine should be triggered. This triggering can be enforced by values equal or below 10.000 that then can be checked in the set routine.
- The builder is *not* exercised when a glue or kern is injected so there can be multiple of
- Adding a box triggers the builder as does the result of an alignment which can be a list of boxes.
- When the output routine is finished the builder is executed because the routine can have pushed back content.
- When a new paragraph is triggered by the `\par` command the builder kicks in but only when the engine was able to enter vertical mode.
- When the job is finished the builder will make sure that pending content is handled.
- An insert and vadjust *can* trigger the builder but only at the nesting level zero which no
- At the beginning of a paragraph (like text), before display math is entered, and when display math ends the builder is also activated.

At the \TeX the builder is triggered automatically in the mentioned cases but at the Lua end

The properties that relate to the page look like counter and dimension registers but they are not. These variables are global and managed differently.

<code>\pagegoal</code>	the available space
<code>\pagetotal</code>	the accumulated space
<code>\pagestretch</code>	the possible zero order stretch
<code>\pagefilstretch</code>	the possible one order stretch
<code>\pagefillstretch</code>	the possible second order stretch
<code>\pagefilllstretch</code>	the possible third order stretch

The page builder

<code>\pageshrink</code>	the possible shrink
<code>\pagedepth</code>	the current page depth
<code>\pagevsize</code>	the initial page goal

When the first content is added to an empty page the `\pagegoal` gets the value of `\vsize` and gets frozen but the value is diminished by the space needed by left over inserts. These inserts are managed via a separate list so they don't interfere with the page that itself of course can have additional inserts. The `\pagevsize` is just a (LuaMetaTeX) status variable.

Another variable is `\deadcycles` that registers the number of times the output routine is called without returning result.

14.3 Inserts

We now come to inserts. In traditional TeX an insert is a data structure that runs on top of

In LuaMetaTeX you can set `\insertmode` to 1 and that is what we do in ConTeXt. In that mode TeXt MkIV we use the range 127 upto 255 in order to avoid a clash with registers. In LMT

A consequence of this approach is that we use dedicated commands to set the insert properties:

<code>\insertdistance</code>	glue	the space before the first instance (on a page)
<code>\insertmultiplier</code>	count	a factor that is used to calculate the height used
<code>\insertlimit</code>	dimen	the maximum amount of space on a page to be taken
<code>\insertpenalty</code>	count	the floating penalty (used when set)
<code>\insertmaxdepth</code>	dimen	the maximum split depth (used when set)
<code>\insertstorage</code>	count	signals that the insert has to be stored for later
<code>\insertheight</code>	dimen	the accumulated height of the inserts so far
<code>\insertdepth</code>	dimen	the current depth of the inserts so far
<code>\insertwidth</code>	dimen	the width of the inserts

These commands take a number and an integer, dimension or glue specification. They can be set and queried but setting the dimensions can have side effects. The accumulated height of the inserts is available in `\insertheights` (which can be set too). The `\insertpenalty` determines the penalty applied when a split is needed.

In the output routine the original TeX variable `\insertpenalties` is a counter that keeps the

The LuaMetaTeX specific storage mode `\insertstoring` variable is explained in the next section.

14.4 Storing

This feature is kind of special and still experimental. When `\insertstoring` is set 1, all inserts that have their storage flag set will be saved. Think of a multi column setup where inserts have to end up in the last column. If there are three columns, the first two will store inserts. Then when the last column is dealt with `\insertstoring` can be set to 2 and that will signal the builder that we will inject the inserts. In both cases, the value of this register will be set to zero so that it doesn't influence further processing.

14.5 Synchronizing

The page builder can be triggered by (for instance) a penalty but you can also use `\pagebound` will trigger the page builder but not leave anything behind. (This is experimental.)

14.6 Migration

Todo, nothing new there, so no hurry.

14.7 Callbacks

Todo, nothing new there, so no hurry.

14.7 Colofon

Author	Hans Hagen
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com contextgarden.net ntg-context@ntg.nl

15 Localboxes



low level

TEX

localboxes

Contents

15.1	Introduction	239
15.2	The basics	239
15.3	The interface	242
15.4	The helpers	247

15.1 Introduction

The `LuaTeX` engine inherited a few features from other engines and adding local boxes to `MetaTeX` specific `\localmiddlebox` primitives. When these primitives are used in vertical mode

The original (Omega) idea was that local boxes are used for repetitive punctuation (like quotes) at the left and/or right end of the lines that make up a paragraph. That means that when these primitives inject nodes they actually introduce states so that a stretch of text can be marked.

When this mechanism was cleaned up in `LuaMetaTeX` I decided to investigate if other usage of `MetaTeX` is not compatible with `LuaTeX`.

This is a preliminary, uncorrected manual.

15.2 The basics

This mechanism uses a mix of setting (pseudo horizontal) box registers that get associated with (positions in a) paragraph. When the lines resulting from breaking the list gets packaged into an horizontal (line) box, the local left and right boxes get prepended and appended to the textual part (inside the left, right and parfills kips and left or right hanging margins). When assigning the current local boxes to the paragraph node(s) references to the pseudo registers are used and the packaging actually copies them. This mix of referencing and copying is somewhat tricky but the engine does it best to hide this for the user.

This mechanism is rather useless when not wrapped into some high level mechanism because by default setting these boxes wipes the existing value. In `LuaMetaTeX` you can act on `TeX`, in `LuaMetaTeX` the local box registers have a linked lists of local boxes tagged by index `TeX` here because there we don't have access. This is why usage as in `LuaTeX` will also work in `MetaTeX`.

This mechanism obeys grouping as is demonstrated in the next three examples. The first example is:

```

\start
  \dorecurse{10}{test #1.1 }
  \localleftbox{\blackrule[width=2em,color=darkred] }
  \dorecurse{20}{test #1.2 }
  \removeunwantedspaces
  \localrightbox{ \blackrule[width=3em,color=darkblue]}
  \dorecurse{20}{test #1.3 }
\stop
  \dorecurse{20}{test #1.4 }
% par ends here

```

The next example differs in a subtle way: watch the `keep` keyword, it makes the setting retain after the group ends.

```

\start
  \start
    \dorecurse{10}{test #1.1 }
    \localleftbox keep {\blackrule[width=2em,color=darkred] }
    \dorecurse{20}{test #1.2 }
    \removeunwantedspaces
    \localrightbox { \blackrule[width=3em,color=darkblue]}
    \dorecurse{20}{test #1.3 }
  \stop
  \dorecurse{20}{test #1.4 }
\stop
% par ends here

```

The third example has two times `keep`. This option is `LuaMetaTeX` specific.

```

\start
  \start
    \dorecurse{10}{test #1.1 }
    \localleftbox keep {\blackrule[width=2em,color=darkred] }
    \dorecurse{20}{test #1.2 }
    \removeunwantedspaces
    \localrightbox keep { \blackrule[width=3em,color=darkblue]}
    \dorecurse{20}{test #1.3 }
  \stop
  \dorecurse{20}{test #1.4 }
\stop
% par ends here

```

test 1.1 test 2.1 test 3.1 test 4.1 test 5.1 test 6.1 test 7.1 test 8.1 test 9.1 test 10.1 test
 ■ 1.2 test 2.2 test 3.2 test 4.2 test 5.2 test 6.2 test 7.2 test 8.2 test 9.2 test 10.2 test
 ■ 11.2 test 12.2 test 13.2 test 14.2 test 15.2 test 16.2 test 17.2 test 18.2 test 19.2
 ■ test 20.2 test 1.3 test 2.3 test 3.3 test 4.3 test 5.3 test 6.3 test 7.3 test 8.3 ■
 ■ test 9.3 test 10.3 test 11.3 test 12.3 test 13.3 test 14.3 test 15.3 test 16.3 ■
 ■ test 17.3 test 18.3 test 19.3 test 20.3 test 1.4 test 2.4 test 3.4 test 4.4 test 5.4 test
 6.4 test 7.4 test 8.4 test 9.4 test 10.4 test 11.4 test 12.4 test 13.4 test 14.4 test 15.4 test
 16.4 test 17.4 test 18.4 test 19.4 test 20.4

Example 1

test 1.1 test 2.1 test 3.1 test 4.1 test 5.1 test 6.1 test 7.1 test 8.1 test 9.1 test 10.1 test
 ■ 1.2 test 2.2 test 3.2 test 4.2 test 5.2 test 6.2 test 7.2 test 8.2 test 9.2 test 10.2 test
 ■ 11.2 test 12.2 test 13.2 test 14.2 test 15.2 test 16.2 test 17.2 test 18.2 test 19.2
 ■ test 20.2 test 1.3 test 2.3 test 3.3 test 4.3 test 5.3 test 6.3 test 7.3 test 8.3 ■
 ■ test 9.3 test 10.3 test 11.3 test 12.3 test 13.3 test 14.3 test 15.3 test 16.3 ■
 ■ test 17.3 test 18.3 test 19.3 test 20.3 test 1.4 test 2.4 test 3.4 test 4.4 test 5.4 test
 6.4 test 7.4 test 8.4 test 9.4 test 10.4 test 11.4 test 12.4 test 13.4 test 14.4 test 15.4 test
 16.4 test 17.4 test 18.4 test 19.4 test 20.4

Example 2

test 1.1 test 2.1 test 3.1 test 4.1 test 5.1 test 6.1 test 7.1 test 8.1 test 9.1 test 10.1 test
 ■ 1.2 test 2.2 test 3.2 test 4.2 test 5.2 test 6.2 test 7.2 test 8.2 test 9.2 test 10.2 test
 ■ 11.2 test 12.2 test 13.2 test 14.2 test 15.2 test 16.2 test 17.2 test 18.2 test 19.2 test
 ■ 20.2 test 1.3 test 2.3 test 3.3 test 4.3 test 5.3 test 6.3 test 7.3 test 8.3 test ■
 ■ 9.3 test 10.3 test 11.3 test 12.3 test 13.3 test 14.3 test 15.3 test 16.3 test ■
 ■ 17.3 test 18.3 test 19.3 test 20.3 test 1.4 test 2.4 test 3.4 test 4.4 test 5.4 ■
 ■ test 6.4 test 7.4 test 8.4 test 9.4 test 10.4 test 11.4 test 12.4 test 13.4 test ■
 ■ 14.4 test 15.4 test 16.4 test 17.4 test 18.4 test 19.4 test 20.4 ■

Example 3

Figure 15.1

One (nasty) side effect is that when you set these boxes ungrouped they are applied to whatever follows, which is why resetting them is built in the relevant parts of Con-TeXt. The next examples are typeset grouped and demonstrate the use of indices:

```
\dorecure{20}{before #1 }
\localleftbox{\bf \darkred L 1 }%
\localleftbox{\bf \darkred L 2 }%
\dorecure{20}{after #1 }
```

before_1_before_2_before_3_before_4_before_5_before_6_before_7_before_8_before_9_before_10_before_11_before_12_before_13_before_14_before_15_before_16_before_17_before_18_before_19_before_20_after_1_after_2_after_3_after_4_after_5_after_6_after_7_after_8_a

Indices can be set for both sides:

```
\dorecurse{5}{\localrightbox index #1{ \bf \darkgreen R #1}}%
\dorecurse{20}{before #1 }
\dorecurse{5}{\localleftbox index #1{\bf \darkred L #1 }}%
\dorecurse{20}{after #1 }
```

before_1_before_2_before_3_before_4_before_5_before_6_before_7_before_8_before_9_before_10_before_11_before_12_before_13_before_14_before_15_before_16_before_17_before_18_before_19_before_20_after_1_after_2_after_3_after_4_after_5_after_6_after_7_after_8_after_9_after_10

We can instruct this mechanism to hook the local box into the main par node by using the `par` keyword. Keep in mind that these local boxes only come into play when the lines are broken, so till then changing them is possible.

```
\dorecurse{3}{\localrightbox index #1{ \bf \darkgreen R #1}}%
\dorecurse{20}{before #1 }
\dorecurse{2}{\localleftbox par index #1{\bf \darkred L #1 }}%
\dorecurse{20}{after #1 }
```

L_1_L_2_before_1_before_2_before_3_before_4_before_5_before_6_before_7_before_8_before_9_before_10_before_11_before_12_before_13_before_14_before_15_before_16_before_17_before_18_before_19_before_20_after_1_after_2_after_3_after_4_after_5_after_6_after_7_after_8_after_9_after_10_after_11_after_12_after_13_after_14_after_15_after_16_after_17_after_18_after_19_after_20

15.3 The interface

The interface described here is experimental.

Because it is hard to foresee if this mechanism will be used at all the ConTeXt interface is

```
\definelocalboxes
[lefttext]
[location=lefttext,width=3em,color=darkblue]
\definelocalboxes
```

```
[lefttextx]
[location=lefttext,width=3em,color=darkblue]
```

```
\definelocalboxes
```

```
[righttext]
[location=righttext,width=3em,color=darkyellow]
```

```
\definelocalboxes
```

```
[righttextx]
[location=righttext,width=3em,color=darkyellow]
```

The_order_of_definition_matters!_Here_the_x_variants_have_a_larger_index_number._There_ can_(currently)_be_at_most_256_indices._The_defined_local_boxes_are_triggered_with_\localbo

```
\startnarrower
```

```
\dorecurse{20}{before #1 }%
\localbox[lefttext]{[L] }%
\localbox[lefttextx]{[LL] }%
\localbox[righttext]{ [RR]}%
\localbox[righttextx]{ [R]}%
\dorecurse{20}{ after #1}%
\stopnarrower
```

Watch_how_we_obey_the_margins:_

```
before_1_before_2_before_3_before_4_before_5_before_6_before_7_before_8_before_
9_before_10_before_11_before_12_before_13_before_14_before_15_before_16_be-
fore_17_before_18_before_19_before_20__after_1_after_2_after_3_after_4_after_5_after_6_after_
```

Here_these_local_boxes_have_dimensions._The_predefined_margin_variants_are_virtual._Here_ we_set_up_the_style_and_color:_

```
\setuplocalboxes
```

```
[leftmargin]
[style=\bs,
color=darkgreen]
```

```
\setuplocalboxes
```

```
[rightmargin]
[style=\bs,
color=darkred]
```

```
\dorecurse{2}{
```

```
\dorecurse{10}{some text #1.##1 }%
KEY#1.1%
```

```

\localmargintext[leftmargin]{L #1.1}%
\localmargintext[rightmargin]{R #1.1}%
\dorecure{10}{some text #1.##1 }%
KEY#1.2%
\localmargintext[leftmargin]{L #1.2}%
\localmargintext[rightmargin]{R #1.2}%
\dorecure{10}{some text #1.##1 }%
\blank
}

```

You can also use `leftedge` and `rightedge` but using them here would put them outside the page.

L_1.2 some_text_1.1_some_text_1.2_some_text_1.3_some_text_1.4_some_text_1.5_some_text_1.6_some_text_1.7_some_text_1.8_some_text_1.9_some_text_1.10_KEY1.1some_text_1.1_R1.2te

L_2.2 some_text_2.1_some_text_2.2_some_text_2.3_some_text_2.4_some_text_2.5_some_text_2.6_some_text_2.7_some_text_2.8_some_text_2.9_some_text_2.10_KEY2.1some_text_2.1_R2.2te

In previous examples you can see that setting something at the left will lag behind so deep down we use another trick here: `\localmiddlebox`. When these boxes get placed a callback can be triggered and in `ConTeXt` we use that to move these middle boxes.

Next we implement line numbers. Watch out: this will not replace the existing mechanisms, it's just an alternative as we have alternative table mechanisms. We have a repertoire of helpers for constructing the result:

```

\definelocalboxes
[linenumberleft]
[command=\LeftNumber,
location=middle,
distance=\leftmargindistance,
width=3em,
style=\bs,
color=darkred]

\definelocalboxes
[linenumberright] % [linenumberleft]
[command=\RightNumber,
location=middle,
distance=\rightmargindistance,
width=3em,

```

```

style=\bf,
color=darkgreen]

\definecounter[MyLineNumberL]
\definecounter[MyLineNumberR]

\setupcounter
  [MyLineNumberL]
  [numberconversion=characters]

\setupcounter
  [MyLineNumberR]
  [numberconversion=romannumerals]

\def\LineNumberL
  {\incrementcounter[MyLineNumberL]%
   \convertedcounter[MyLineNumberL]}

\def\LineNumberR
  {\incrementcounter[MyLineNumberR]%
   \convertedcounter[MyLineNumberR]}

\protected\def\LeftNumber
  {\setbox\localboxcontentbox\hbox
   to \localboxesparameter{width}
   {(\LineNumberL\hss\strut)}%
   \localmarginlefttext\zeropoint}

\protected\def\RightNumber
  {\setbox\localboxcontentbox\hbox
   to \localboxesparameter{width}
   {(\strut\hss\LineNumberR)}%
   \localmarginrighttext\zeropoint}

\localbox[linenumberleft]{}%
\localbox[linenumberright]{}%
\dorecurse{2}{
  \samplefile{tufte}
  \par
}
\resetlocalbox[linenumberleft]%
\resetlocalbox[linenumberright]%

```

We_use_our_tufte_example_to_illustrate_the_usage:_

(15.d) We_thrive_in_information-thick_worlds_because_of_our_marvelous_and_everyday_capac(15.i)
 (15.b) ity_to_select, edit, single_out, structure, highlight, group, pair, merge, harmonize, syn(15.ii)
 (15.d) thesize, focus, organize, condense, reduce, boil_down, choose, categorize, catalog, cla(15.iii)
 (15.d) sify, list, abstract, scan, look_into, idealize, isolate, discriminate, distinguish, screen, (15.iv)
 (15.e) geonhole, pick_over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, (15.v)
 (15.f) erage, approximate, cluster, aggregate, outline, summarize, itemize, review, dip_into, (15.vi)
 (15.g) through, browse, glance_into, leaf_through, skim, refine, enumerate, glean, synopsized(15.vii)
 (15.h) now_the_wheat_from_the_chaff_and_separate_the_sheep_from_the_goats.__ (15.viii)

(15.i) We_thrive_in_information-thick_worlds_because_of_our_marvelous_and_everyday_capac(15.ix)
 (15.j) ity_to_select, edit, single_out, structure, highlight, group, pair, merge, harmonize, syn(15.x)
 (15.k) thesize, focus, organize, condense, reduce, boil_down, choose, categorize, catalog, cla(15.xi)
 (15.l) sify, list, abstract, scan, look_into, idealize, isolate, discriminate, distinguish, screen, (15.xii)
 (15.m) geonhole, pick_over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, (15.xiii)
 (15.n) erage, approximate, cluster, aggregate, outline, summarize, itemize, review, dip_into, (15.xiv)
 (15.o) through, browse, glance_into, leaf_through, skim, refine, enumerate, glean, synopsized(15.xv)
 (15.p) now_the_wheat_from_the_chaff_and_separate_the_sheep_from_the_goats.__ (15.xvi)

For_convenience_we_support_ranges_like_this_(we've_reset_the_line_number_counters_
 here):_

```
\startlocalboxrange[linenumberleft]%
\startlocalboxrange[linenumberright]%
\dorecurse{2}{
  \samplefile{tufte}
  \par
}
\stoplocalboxrange
\stoplocalboxrange
```

(15.d) We_thrive_in_information-thick_worlds_because_of_our_marvelous_and_everyday_capac(15.i)
 (15.b) ity_to_select, edit, single_out, structure, highlight, group, pair, merge, harmonize, syn(15.ii)
 (15.d) thesize, focus, organize, condense, reduce, boil_down, choose, categorize, catalog, cla(15.iii)
 (15.d) sify, list, abstract, scan, look_into, idealize, isolate, discriminate, distinguish, screen, (15.iv)
 (15.e) geonhole, pick_over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, (15.v)
 (15.f) erage, approximate, cluster, aggregate, outline, summarize, itemize, review, dip_into, (15.vi)
 (15.g) through, browse, glance_into, leaf_through, skim, refine, enumerate, glean, synopsized(15.vii)
 (15.h) now_the_wheat_from_the_chaff_and_separate_the_sheep_from_the_goats.__ (15.viii)

(15.i) We thrive in information-thick worlds because of our marvelous and everyday capacities to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, geonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, erage, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsized now the wheat from the chaff and separate the sheep from the goats.

15.4 The helpers

For the moment we have these helpers:

<code>\localboxindex</code>	integer
<code>\localboxlinenumber</code>	integer
	<code>\localboxlinewidth</code> dimension
<code>\localboxlocalwidth</code>	dimension
<code>\localboxprogress</code>	dimension
<code>\localboxleftoffset</code>	dimension
<code>\localboxrightoffset</code>	dimension
	<code>\localboxleftskip</code> dimension
<code>\localboxrightskip</code>	dimension
<code>\localboxlefthang</code>	dimension
<code>\localboxrighthang</code>	dimension
	<code>\localboxindent</code> dimension
<code>\localboxparfillleftskip</code>	dimension
<code>\localboxparfillrightskip</code>	dimension
<code>\localboxovershoot</code>	dimension

The progress and offsets are accumulated values of the normalized indent, hangs, skips etc. The line number is the position in the paragraph. In the callback we set the box register `\localboxcontentbox` and use it after the command has been applied. In the line number example you can see how we set its final content, so these boxes are sort of dynamic. Normally in the middle case no content is passed and in the par_builder a middle is not taken into account when calculating the line width.

15.4 Colofon

Author	Hans Hagen
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com contextgarden.net ntg-context@ntg.nl

16 Loops



low level

TEX

loops

Contents

16.1	Introduction	250
16.2	Primitives	250
16.3	Wrappers	254
16.4	About quitting	256
16.5	Simple repeaters	256
16.6	Endless loops	256
16.7	Loop variables	256

16.1 Introduction

I have hesitated long before I finally decided to implement native loops in LuaMeta-TeX. Among the reasons against such a feature is that one can define macros that do loops. In MetaTeX it is actually possible to use the local control mechanism to hide loop counter advances.

In the next sections we describe the new native loop primitives in LuaMetaTeX as well as the TeXt loop helpers.

16.2 Primitives

Because MetaPost, which is also a macro language, has native loops, it makes sense to also have native loops in TeX and in LuaMetaTeX it was not that hard to add it. One variant

```
\localcontrolledloop 1 100000 1 {%
  % body
}
```

Here is an example of usage:

```
\localcontrolledloop 1 5 1 {%
  [\number\currentloopiterator]
  \localcontrolledloop 1 10 1 {%
    (\number\currentloopiterator)
  }%
  [\number\currentloopiterator]
  \par
}
```

The `\currentloopiterator` is a numeric token so you need to explicitly serialize it with `\number` or `\the` if you want it to be typeset.

```
[1]_(1)_(2)_(3)_(4)_(5)_(6)_(7)_(8)_(9)_(10)_[1]_
[2]_(1)_(2)_(3)_(4)_(5)_(6)_(7)_(8)_(9)_(10)_[2]_
[3]_(1)_(2)_(3)_(4)_(5)_(6)_(7)_(8)_(9)_(10)_[3]_
[4]_(1)_(2)_(3)_(4)_(5)_(6)_(7)_(8)_(9)_(10)_[4]_
[5]_(1)_(2)_(3)_(4)_(5)_(6)_(7)_(8)_(9)_(10)_[5]_
```

Here_is_another_example.This_time_we_also_show_the_current_nesting:

```
\localcontrolledloop 1 100 1 {%
  \ifnum\currentloopiterator>6\relax
    \quitloop
  \else
    [\number\currentloopnesting:\number\currentloopiterator]
    \localcontrolledloop 1 8 1 {%
      (\number\currentloopnesting:\number\currentloopiterator)
    }\par
  \fi
}
```

Watch_the_\quitloop: it_will_end_the_loop_at_the_next_iteration_so_any_content_after_it_will_s

```
[1:1]_(2:1)_(2:2)_(2:3)_(2:4)_(2:5)_(2:6)_(2:7)_(2:8)_
[1:2]_(2:1)_(2:2)_(2:3)_(2:4)_(2:5)_(2:6)_(2:7)_(2:8)_
[1:3]_(2:1)_(2:2)_(2:3)_(2:4)_(2:5)_(2:6)_(2:7)_(2:8)_
[1:4]_(2:1)_(2:2)_(2:3)_(2:4)_(2:5)_(2:6)_(2:7)_(2:8)_
[1:5]_(2:1)_(2:2)_(2:3)_(2:4)_(2:5)_(2:6)_(2:7)_(2:8)_
[1:6]_(2:1)_(2:2)_(2:3)_(2:4)_(2:5)_(2:6)_(2:7)_(2:8)_
```

The_three_loop_variants_all_perform_differently:_

```
l:\testfeatureonce {1000} {\localcontrolledloop 1 2000 1 {\relax}} %
  \elapsedtime
e:\testfeatureonce {1000} {\expandedloop 1 2000 1 {\relax}} %
  \elapsedtime
u:\testfeatureonce {1000} {\unexpandedloop 1 2000 1 {\relax}} %
  \elapsedtime
```

An_unexpanded_loop_is_(of_course)_the_fastest_because_it_only_collects_and_then_feeds_back_the_lot.In_an_expanded_loop_each_cycle_does_an_expansion_of_the_body_and_collects_the_result_which_is_then_injected_afterwards_and_the_controlled_loop_just_expands_the_body_each_iteration._

```
l:_0.114
```

```
e:_0.101
u:_0.031
```

The different behavior is best illustrated with the following example:

```
\edef\TestA{\localcontrolledloop 1 5 1 {A}} % out of order
\edef\TestB{\expandedloop 1 5 1 {B}}
\edef\TestC{\unexpandedloop 1 5 1 {C\relax}}
```

We can show the effective definition:

```
\meaningasis\TestA
\meaningasis\TestB
\meaningasis\TestC
```

```
A: \TestA
B: \TestB
C: \TestC
```

Watch how the first test pushes the content in the main input stream:

```
AAAAA_
\def\_TestA_{ }
\def\_TestB_{BBBBB}
\def\_TestC_{C\relax_C\relax_C\relax_C\relax_C\relax_}

A:_
B:_BBBBB
C:_CCCCC
```

Here are some examples that show what gets expanded and what not:

```
\edef\whatever
  {\expandedloop 1 10 1
   {(\number\currentloopiterator)
    \scratchcounter=\number\currentloopiterator\relax}}
```

```
\meaningasis\whatever
```

```
\def\_whatever_{(1)\scratchcounter=1\relax(2)\scratchcounter=2\relax_
(3)\scratchcounter=3\relax(4)\scratchcounter=4\relax(5)\scratchcounter=5\relax_
(6)\scratchcounter=6\relax(7)\scratchcounter=7\relax(8)\scratchcounter=8\relax_
(9)\scratchcounter=9\relax(10)\scratchcounter=10\relax_}
```

A_local_control_encapsulation_hides_the_assignment:_

```
\edef\whatever
  {\expandedloop 1 10 1
    {(\number\currentloopiterator)
      \beginlocalcontrol
        \scratchcounter=\number\currentloopiterator\relax
      \endlocalcontrol}}
```

```
\meaningasis\whatever
```

```
\def\whatever_{(1)_(2)_(3)_(4)_(5)_(6)_(7)_(8)_(9)_(10)_}
```

Here_we_see_the_assignment_being_retained_but_with_changing_values:_

```
\edef\whatever
  {\unexpandedloop 1 10 1
    {\scratchcounter=1\relax}}
```

```
\meaningasis\whatever
```

```
\def\whatever_{\scratchcounter=1\relax\scratchcounter=1\relax\scratchcounter_
=1\relax\scratchcounter=1\relax\scratchcounter=1\relax\scratchcounter=1\relax_
\scratchcounter=1\relax\scratchcounter=1\relax\scratchcounter=1\relax_
\scratchcounter=1\relax_}
```

We_get_no_expansion_at_all:_

```
\edef\whatever
  {\unexpandedloop 1 10 1
    {\scratchcounter=\the\currentloopiterator\relax}}
```

```
\meaningasis\whatever
```

```
\def\whatever_{\scratchcounter=0\relax\scratchcounter=0\relax\scratchcounter_
=0\relax\scratchcounter=0\relax\scratchcounter=0\relax\scratchcounter=0\relax_
\scratchcounter=0\relax\scratchcounter=0\relax\scratchcounter=0\relax_
\scratchcounter=0\relax_}
```

And_here_we_have_a_mix:_

```
\edef\whatever
  {\expandedloop 1 10 1
    {\scratchcounter=\the\currentloopiterator\relax}}
```

`\meaningasis\whatever`

```
\def\whatever_{\scratchcounter=1\relax\scratchcounter=2\relax\scratchcounter=
=3\relax\scratchcounter=4\relax\scratchcounter=5\relax\scratchcounter=6\relax
\scratchcounter=7\relax\scratchcounter=8\relax\scratchcounter=9\relax
\scratchcounter=10\relax}
```

There_is_one_feature_worth_noting. When_you_feed_three_numbers_in_a_row_like_here, there_is_a_danger_of_them_being_seen_as_one:_

```
\expandedloop
  \number\dimexpr1pt
  \number\dimexpr2pt
  \number\dimexpr1pt
  {}
```

This_gives_an_error_because_a_too_large_number_is_seen. Therefore, these loops permit_leading_equal_signs, as_in_assignments (we could support_keywords_but_it_doesn't_make_much_sense):_

```
\expandedloop =\number\dimexpr1pt =\number\dimexpr2pt =\number\dimexpr1pt{}
```

16.3 Wrappers

We_always_had_loop_helpers_in_ConT_EXt_and_the_question_is: “What_we_will_gain_when_we_replace_the_definitions_with_ones_using_the_above?”. The_answer_is: “We_have_little_performance_but_not_as_much_as_one_expects!”. This_has_to_do_with_the_fact_that_we_support_#1_as_iterator_and_#2_as (verbose) nesting_values_and_that_comes_with_some_overhead. It_is_also_the_reason_why_these_loop_macros_are_protected (unexpandable). However, using_the_primitives_might_look_somewhat_more_natural_in_low_level_T_EX_code._

Also, replacing_their_definitions_can_have_side_effects_because_the_primitives_are (and_will_be) still_experimental_so_it's typically_a_patch_that_I_will_run_on_my_machine_for_a_while._

Here_is_an_example_of_two_loops. The_inner_state_variables_have_one_hash, the_outer_one_extra:_

```
\dorecurse{2}{
  \dostepwiserecurse{1}{10}{2}{
    (#1:#2) [##1:##2]
```

```

    }\par
}

```

We_get_this:_

```

(1:1)_[1:2]_(1:1)_[3:2]_(1:1)_[5:2]_(1:1)_[7:2]_(1:1)_[9:2]_
(2:1)_[1:2]_(2:1)_[3:2]_(2:1)_[5:2]_(2:1)_[7:2]_(2:1)_[9:2]_

```

We_can_also_use_two_state_macro_but_here_we_would_have_to_store_the_outer_ones:_

```

\dorecuse {2} {
    /\recursedepth:\recurselevel/
    \dostepwiserecuse {1} {10} {2} {
        <\recursedepth:\recurselevel>
    }\par
}

```

That_gives_us:_

```

/1:1/_<2:1>_<2:3>_<2:5>_<2:7>_<2:9>_
/1:2/_<2:1>_<2:3>_<2:5>_<2:7>_<2:9>_

```

An_endless_loop_works_as_follows:_

```

\doloop {
    ...
    \ifsomeconditionismet
        ...
        \exitloop
    \else
        ...
    \fi
    % \exitloopnow
    ...
}

```

Because_of_the_way_we_quit_there_will_not_be_a_new_implementation_in_terms_of_the_loop_primitives. You_need_to_make_sure_that_you_don't_leave_in_the_middle_of_an_on-going_condition. The_second_exit_is_immediate._

We_also_have_a_(simple)_expanded_variant:_

```

\edef\TestX{\doexpandedrecurse{10}{!}} \meaningasis\TestX

```

This helper can be implemented in terms of the loop primitives which makes them a bit faster, but these are not critical:

```
\def\TestX_{!!!!!!!}
```

A variant that supports #1 is the following:

```
\edef\TestX{\doexpandedrecursed{10}{#1}} \meaningasis\TestX
```

So:

```
\def\TestX_{12345678910}
```

16.4 About quitting

You can quit a local and expanded loop at the next iteration using `\quitloop`. With `\quitloopnow` you immediately leave the loop but you need to beware of side effects, like not ending a condition properly. Keep in mind that a macro language like `TEX` is not that fr

16.5 Simple repeaters

For simple iterations we have `\localcontrolledrepeat`, `\expandedrepeat`, `\unexpandedrepeat`. These take one integer instead of three: the final iterator value.

16.6 Endless loops

There are three endless loop primitives: `\localcontrolledendless`, `\expandedendless`, `\unexpandedendless`. These will keep running till you quit them. The loop counter can overflow the maximum integer value and will then start again at 1.

16.7 Loop variables

The following example shows how we can access the current, parent and grand parent loop iterator values using a parameter like syntax:

```
\localcontrolledloop 1 4 1 {%
  \localcontrolledloop 1 3 1 {%
    \localcontrolledloop 1 2 1 {%
      \edef\foo{[#G,#P,#I]}\foo
    \def \oof{<#G,#P,#I>}\oof
```

```

        (#G,#P,#I)\space
    }
\par
}
}

```

```

[1,1,1]<1,1,1>(1,1,1)_[1,1,2]<1,1,2>(1,1,2)___
[1,2,1]<1,2,1>(1,2,1)_[1,2,2]<1,2,2>(1,2,2)___
[1,3,1]<1,3,1>(1,3,1)_[1,3,2]<1,3,2>(1,3,2)___
[2,1,1]<2,1,1>(2,1,1)_[2,1,2]<2,1,2>(2,1,2)___
[2,2,1]<2,2,1>(2,2,1)_[2,2,2]<2,2,2>(2,2,2)___
[2,3,1]<2,3,1>(2,3,1)_[2,3,2]<2,3,2>(2,3,2)___
[3,1,1]<3,1,1>(3,1,1)_[3,1,2]<3,1,2>(3,1,2)___
[3,2,1]<3,2,1>(3,2,1)_[3,2,2]<3,2,2>(3,2,2)___
[3,3,1]<3,3,1>(3,3,1)_[3,3,2]<3,3,2>(3,3,2)___
[4,1,1]<4,1,1>(4,1,1)_[4,1,2]<4,1,2>(4,1,2)___
[4,2,1]<4,2,1>(4,2,1)_[4,2,2]<4,2,2>(4,2,2)___
[4,3,1]<4,3,1>(4,3,1)_[4,3,2]<4,3,2>(4,3,2)___

```

Another_way_to_access_a(ny)_parent_is:_

```

\localcontrolledloop 1 4 1 {%
    \localcontrolledloop 1 3 1 {%
        \localcontrolledloop 1 2 1 {%
            (\the\previousloopiterator2,%
            \the\previousloopiterator1,%
            \the\currentloopiterator)
        }
    \par
    }
}

```

These_methods_make_that_one_doesn't_have_to_store_the_outer_loop_variables_for_usage_inside_the_inner_loop._Watch_out_with_the_\edef:_

```

\edef\foo{[#G,#P,#I]}
\def \oof{<#G,#P,#I>}

```

```

\localcontrolledloop 1 4 1 {%
    \localcontrolledloop 1 3 1 {%
        \localcontrolledloop 1 2 1 {%
            %

```

```

% I iterator      \currentloopiterator
% P parent        \previousloopiterator1
% G grandparent   \previousloopiterator2
%
\edef\of{[#G,#P,#I]}%
\foo\oof\of{[#G,#P,#I]}\space
%
}
\par
}
}

[0,0,0]<1,1,1>[1,1,1](1,1,1)_ [0,0,0]<1,1,2>[1,1,2](1,1,2)_
[0,0,0]<1,2,1>[1,2,1](1,2,1)_ [0,0,0]<1,2,2>[1,2,2](1,2,2)_
[0,0,0]<1,3,1>[1,3,1](1,3,1)_ [0,0,0]<1,3,2>[1,3,2](1,3,2)_
[0,0,0]<2,1,1>[2,1,1](2,1,1)_ [0,0,0]<2,1,2>[2,1,2](2,1,2)_
[0,0,0]<2,2,1>[2,2,1](2,2,1)_ [0,0,0]<2,2,2>[2,2,2](2,2,2)_
[0,0,0]<2,3,1>[2,3,1](2,3,1)_ [0,0,0]<2,3,2>[2,3,2](2,3,2)_
[0,0,0]<3,1,1>[3,1,1](3,1,1)_ [0,0,0]<3,1,2>[3,1,2](3,1,2)_
[0,0,0]<3,2,1>[3,2,1](3,2,1)_ [0,0,0]<3,2,2>[3,2,2](3,2,2)_
[0,0,0]<3,3,1>[3,3,1](3,3,1)_ [0,0,0]<3,3,2>[3,3,2](3,3,2)_
[0,0,0]<4,1,1>[4,1,1](4,1,1)_ [0,0,0]<4,1,2>[4,1,2](4,1,2)_
[0,0,0]<4,2,1>[4,2,1](4,2,1)_ [0,0,0]<4,2,2>[4,2,2](4,2,2)_
[0,0,0]<4,3,1>[4,3,1](4,3,1)_ [0,0,0]<4,3,2>[4,3,2](4,3,2)_

```

16.7 Colofon

Author	Hans Hagen
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com contextgarden.net ntg-context@ntg.nl

17 Tokens

low level

TEX

tokens

Contents

17.1	Introduction	260
17.2	What are tokens	260
17.3	Some implementation details	263
17.4	Other data management	263
17.5	Macros	264
17.6	Looking at tokens	264

17.1 Introduction

Most users don't need to know anything about tokens but it happens that when \TeX ies meet

That said, because in documents about \TeX the word ‘token’ does pop up I will try to give a little insight here. But for using \TeX it's mostly irrelevant. The descriptions below_kens as I see them.”

17.2 What are tokens

Both the words ‘node’ and ‘token’ are quite common in programming and also rather old which is proven by the fact that they also are used in the \TeX source. A node is a ‘storage character’ nodes (or in \LaTeX speak ‘glyph’ nodes) with properties like the font and the character referred to. But before that happens, the three characters in the input `t`, `e` and `x` interpreted as in this case being just that: characters. When you enter `\TeX` the input processors first sees a backslash and because that has a special meaning in \TeX it will you enter `$` \TeX will look ahead for a second one in order to determine display math, push

A token is internally just a 32 bit number that encodes what \TeX has seen. It is the assembly our example gets tagged as such and encoded in this number in a way that the intention can be derived later on.

Now, the way \TeX looks at these tokens can differ. In some cases it will just look at this (32

token	=	cmd	chr
-------	---	-----	-----

²³ Talking about fashion: it would be more impressive to talk about \TeX and friends as a software stack than calling it a distribution. Today, it's all about marketing.

Back to the three characters: these become tokens where the command code indicates that it is a letter and the char code stores what letter we have at hand and in the case of LuaTeX and LuaMetaTeX these are Unicode values. Contrary to the traditional code engines an utf sequence is read, but these multiple bytes still become one number that code has plenty of characters slots you can imagine that combining 16 catcode commands with code values makes a large repertoire of tokens.

There are more commands than the 16 basic characters related ones, in LuaMetaTeX we have just over 150 command codes (LuaTeX has a few more but they are also organized in MetaTeX the first one has sub command code 9 (vbox code) and the second one has code 1).

Now, before we move on it is important to know that all these codes are in fact abstract numbers. Although it is quite likely that engines that are derived from each other have similar numbers (just more) this is not the case for LuaMetaTeX. Because the input (TeX) the command and char codes have been reorganized in a such a way that exposure is (TeXt LMTX and LuaMetaTeX were pretty useable during the process), but also had to (re)code.

So, input is converted into tokens, in most cases one-by-one. When a token is assembled, it either gets stored (deliberately or as part of some look ahead scanning), or it immediately gets (what is called:) expanded. Depending on what the command is, some action is triggered. For instance, a character gets appended to the node list immediately. An `\hbox` command will start assembling a box which its own node list that then gets some treatment: if this primitive was a follow up on `\setbox` it will get stored, otherwise it might end up in the current node list as so called hlist node. Commands that relate to registers have `0xFFFF` char codes because that is how many registers we have per category.

When a token gets stored for later processing it becomes part of a larger data structure, a so called memory word. These memory words are taken from a large pool of words and they store a token and additional properties. The info field contains the token value, the mentioned command and char. When there is no linked list, the link can actually be used to store a value, something that in LuaMetaTeX we actually do.

1	info	link
2	info	link
3	info	link
n	info	link

When for instance we say `\toks 0 {tex}` the scanner sees an escape, followed by 4 letters (toks) and the escape triggers a lookup of the primitive (or macro or ...) with that 0 will push back a copy of this list into the input.

What are tokens

In addition to the token memory pool, there is also a table of equivalents. That one is part of a larger table of memory words where TeX stores all it needs to store. The 16 groups

main hash	null control sequence
	128K hash entries
	frozen control sequences
	special sequences (undefined)
registers	17 internal & 64K user glues
	4 internal & 64K user mu glues
	12 internal & 64K user tokens
	2 internal & 64K user boxes
	116 internal & 64K user integers
	0 internal & 64K user attribute
	22 internal & 64K user dimensions
specifications	5 internal & 0 user
extra hash	additional entries (grows dynamic)

So, a letter token `t` is just that, a token. A token referring to a register is again just a number, but its char code points to a slot in the equivalents table. A macro, which we haven't discussed yet, is actually just a token list. When a name lookup happens the hash table is consulted and that one runs in parallel to part of the table of equivalents. When there is a match, the corresponding entry in the equivalents table points to a token list.

1	string index	equivalents or (next > n) index
2	string index	equivalents or (next > n) index
n	string index	equivalents or (next > n) index
n + 1	string index	equivalents or (next > n) index
n + 2	string index	equivalents or (next > n) index
n + m	string index	equivalents or (next > n) index

It sounds complex and it actually also is somewhat complex. It is not made easier by the fact that we also track information related to grouping (saving and restoring), need reference counts for copies of macros and token lists, sometimes store information directly instead of via links to token lists, etc. And again one cannot compare Lua-MetaTeX with the other engines. Because we did away with some of the limitations of the t

What are tokens

1	level	type	flag	value
2	level	type	flag	value
3	level	type	flag	value
n	level	type	flag	value

So, here `LuaMetaTeX` differs from other engines because it combines two tables, which is possible in `MetaTeX` macros can have extra properties (flags) and these also need one byte. Contrary to

Because a macro starts with a reference count we have some room in the info field to store information about it having arguments or not. It is these details that make `LuaMetaTeX` a bit more efficient in terms of memory usage and performance than its ancestor `TeX`. But as with the other changes, it was a very stepwise process in order to keep the sys

17.3 Some implementation details

Sometimes there is a special head token at the start. This makes for easier appending of extra tokens. In traditional `TeX` node lists are forward linked, in `LuaTeX` they are dou

For various reasons original `TeX` uses global variables temporary lists. This is for instance not in `TeX` we often just serialize lists and using local variables makes more sense. One of the first changes in `MetaTeX` was to group all global variables in (still global) structures but well isolated. That

Because `TeX` had to run on machines that we nowadays consider rather limited, it had to be efficient. In `MetaTeX` we stay as close to original `TeX` as possible but there have been some improvements. In `TeX` we store much more in nodes (each has a prev pointer and an attribute list pointer and

17.4 Other data management

There is plenty going on in `TeX` when it processes your input, just to mention a few:

- Grouping is handled by a nesting stack.
- Nested conditionals (`\if...`) have their own stack.
- The values before assignments are saved on the save stack.
- Also other local changes (housekeeping) ends up in the save stack.
- Token lists and macro aliases have references pointers (reuse).
- Attributes, being linked node lists, have their own management.

²⁴ On the agenda of `LuaMetaTeX` is to use this property in the underlying code, that doesn't yet profit from this and therefore keep previous pointers in store.

In all these subsystems tokens or references to tokens can play a role. Reading a single character from the input can trigger a lot of action. A curly brace tagged as `begin_group` command will push the grouping level and from then on registers and some other quantities that are changed will be stored on the save stack so that after the group ends they can be restored. When primitives take keywords, and `no_match` happens, tokens are pushed back into the input which introduces a new input level (also some stack). When numbers are read a token that represents no digit is pushed back too and macro packages use numbers and dimensions a lot. It is a surprise that \TeX is so fast.

17.5 Macros

There is a distinction between primitives, the build in commands, and macros, the commands defined by users. A primitive relates to a command code and char code but macros are, unless they are made an alias to something else, like a `\countdef` or `\let`, sically pointers to a token list. There is some additional data stored that makes it possible to parse and grab arguments.

When we have a control sequence (macro) `\controlsequence` the name is looked up in the hash table. When found its value will point to the table of equivalents. As mentioned, that table keeps track of the cmd and points to a token list (the meaning). We saw that this table also stores the current level of grouping and flags.

If we say, in the input, `\hbox to 10pt {x\hss}`, the box is assembled as we go and when it is appended to the current node list there are no tokens left. When scanning this, the engine literally sees a backslash and the four letters `hbox`. However when we have this:

```
\def\MyMacro{\hbox to 10pt {x\hss}}
```

the `\hbox` has become one memory word which has a token representing the `\hbox` primitive plus a link to the next token. The space after a control sequence is gobbled so the next two tokens, again stored in a linked memory word, are letter tokens, followed by two other and two letter tokens for the dimensions. Then we have a space, a brace, a letter, a primitive and a brace. The about 20 characters in the input became a dozen memory words each two times four bytes, so in terms of memory usage we end up with quite a bit more. However, when \TeX runs over that list it only has to interpret the tol

17.6 Looking at tokens

When you say `\tracingall` you will see what the engine does: read input, expand primitives and macros, typesetting etc. You might need to set `\tracingonline` to get

a_bit_more_output_on_the_console._One_way_to_look_at_macros_is_to_use_the_\meaning_command,_so_if_we_have:_

```
\permanent\protected\def\MyMacro#1#2{Do #1 or #2!}
```

we_can_say_this:_

```
\meaning      \MyMacro
```

```
\meaningless\MyMacro
```

```
\meaningfull\MyMacro
```

and_get:_

```
protected_macro:#1#2->Do #1_or_#2!
```

```
#1#2->Do #1_or_#2!
```

```
permanent_protected_macro:#1#2->Do #1_or_#2!
```

You_get_less_when_you_ask_for_the_meaning_of_a_primitive,_just_its_name._The_\meaningfull_primitive_gives_the_most_information._In_LuaMetaTeX_protected_macros_are_first_class_commands:_MetaTeX._Instead_we_have_\tolerant_macros_but_that's_another_story._The_flags_that_were_n

For_the_above_definition,_the_\showluatokens_command_will_show_a_meaning_on_the_console._

```
\showluatokens\MyMacro
```

This_gives_the_next_list,_where_the_first_column_is_the_address_of_the_token,_the_second_one_the_command_code,_and_the_third_one_the_char_code._When_there_are_arguments_involved,_the_list_of_what_needs_to_get_matched_is_shown._

```
permanent protected control sequence: MyMacro
501263  19   49  match                argument 1
501087  19   50  match                argument 2
385528  20    0  end match
-----
501090  11   68  letter                D (U+00044)
 30833  11  111  letter                o (U+0006F)
500776  10   32  spacer
385540  21    1  parameter reference
112057  10   32  spacer
431886  11  111  letter                o (U+0006F)
 30830  11  114  letter                r (U+00072)
 30805  10   32  spacer
```

```
500787 21    2 parameter reference
213412 12   33 other char          ! (U+00021)
```

In the next subsections I will give some examples. This time we use helper defined in a module:

```
\usemodule[system-tokens]
```

17.6.1 Example 1: in the input

```
\luatokenable{1 \bf{2} 3\what {!}}
```

given token list:

<no tokens>

17.6.2 Example 2: in the input

```
\luatokenable{a \the\scratchcounter b \the\parindent \hbox to 10pt{x}}
```

given token list:

<no tokens>

17.6.3 Example 3: user registers

```
\scratchtoks{foo \framed{\red 123}456}
```

```
\luatokenable\scratchtoks
```

token register: scratchtoks

<no tokens>

17.6.4 Example 4: internal variables

```
\luatokenable\everypar
```

internal token variable: everypar

<no tokens>

17.6.5 Example 5: macro definitions

```
\protected\def\whatever#1[#2](#3)\relax
  {oops #1 and #2 & #3 done ## error}

\luatokenable\whatever
```

protected control sequence: whatever					
598203	19	49	match		argument 1
597245	12	91	other char	[U+0005B	
595910	19	50	match		argument 2
598255	12	93	other char] U+0005D	
594255	12	40	other char	(U+00028	
587522	19	51	match		argument 3
597801	12	41	other char) U+00029	
593614	16	0	relax		relax
596384	20	0	end match		
594803	11	111	letter	o U+0006F	
594684	11	101	letter	e U+00065	
597913	11	112	letter	p U+00070	
593875	11	115	letter	s U+00073	
598132	10	32	spacer		
596786	21	1	parameter reference		
595954	10	32	spacer		
596521	11	97	letter	a U+00061	
596021	11	110	letter	n U+0006E	
598026	11	100	letter	d U+00064	
598299	10	32	spacer		
598160	21	2	parameter reference		
594020	10	32	spacer		
596070	12	38	other char	& U+00026	
597418	10	32	spacer		
598275	21	3	parameter reference		
597764	10	32	spacer		
595400	11	100	letter	d U+00064	
600241	11	111	letter	o U+0006F	
598328	11	110	letter	n U+0006E	
597447	11	101	letter	e U+00065	
599638	10	32	spacer		
597231	6	35	parameter		
598706	10	32	spacer		
600276	11	101	letter	e U+00065	
596590	11	114	letter	r U+00072	
597207	11	114	letter	r U+00072	
595440	11	111	letter	o U+0006F	
596080	11	114	letter	r U+00072	

17.6.6 Example 6: commands

```
\luatokenable\startitemize
```

\luatokentable\stopitemize

frozen instance protected control sequence: startitemize					
520953	147	0	tolerant protected call		startitemgroup
520954	12	91	other char	[U+0005B	
520955	11	105	letter	i U+00069	
520956	11	116	letter	t U+00074	
520957	11	101	letter	e U+00065	
520958	11	109	letter	m U+0006D	
520959	11	105	letter	i U+00069	
520960	11	122	letter	z U+0007A	
520961	11	101	letter	e U+00065	
520962	12	93	other char] U+0005D	

frozen instance protected control sequence: stopitemize

431271	143	0	protected call	stopitemgroup
--------	-----	---	----------------	---------------

17.6.7 Example 7: commands**\luatokentable\doifelse**

permanent protected control sequence: doifelse					
55400	19	49	match	argument 1	
55401	19	50	match	argument 2	
55402	20	0	end match		
55403	137	29	if test	iftok	
55404	1	123	left brace		
55405	21	1	parameter reference		
55406	2	125	right brace		
55407	1	123	left brace		
55408	21	2	parameter reference		
55409	2	125	right brace		
55410	131	0	expand after	expandafter	
55411	142	0	call	firstoftwoarguments	
55412	137	3	if test	else	
55413	131	0	expand after	expandafter	
55414	142	0	call	secondoftwoarguments	
55415	137	2	if test	fi	

17.6.8 Example 8: nothing**\luatokentable\relax**

primitive control sequence: relax					
<no tokens>					

17.6.9 Example 9: hashes

```
\edef\foo#1#2{(#1)(\letterhash)(#2)} \luatokenable\foo
```

control sequence: foo					
596021	19	49	match		argument 1
597820	19	50	match		argument 2
595444	20	0	end match		
596193	12	40	other char	(U+00028
596219	21	1	parameter reference		
599325	12	41	other char)	U+00029
598486	12	40	other char	(U+00028
599258	12	35	other char	#	U+00023
596734	12	41	other char)	U+00029
595426	12	40	other char	(U+00028
597426	21	2	parameter reference		
594456	12	41	other char)	U+00029

17.6.10 Example 10: nesting

```
\def\foo#1{\def\foo##1{(#1)(##1)}} \luatokenable\foo
```

control sequence: foo					
599921	19	49	match		argument 1
598612	20	0	end match		
598645	128	1	def		def
595592	142	0	call		foo
594803	6	35	parameter		
595879	12	49	other char	1	U+00031
596874	1	123	left brace		
598299	12	40	other char	(U+00028
599222	21	1	parameter reference		
595569	12	41	other char)	U+00029
471942	12	40	other char	(U+00028
599178	6	35	parameter		
599432	12	49	other char	1	U+00031
600241	12	41	other char)	U+00029
598279	2	125	right brace		

17.6.11 Remark

In all these examples the numbers are to be seen as abstractions. Some command codes and sub-command codes might change as the engine evolves. This is why the LuaMetaTeX engine has lots of Lua functions that provide information about what number r

17.6.11 Colofon

Author	Hans Hagen
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com contextgarden.net ntg-context@ntg.nl

18 Buffers

low level

TEX

buffers

Contents

18.1	Preamble	272
18.2	Encoding	272
18.3	Performance	273
18.4	Files	274
18.5	Macros	274
18.6	Token lists	275
18.7	Buffers	275
18.8	Setups	278
18.9	xml	279
18.10	Lua	279
18.11	Protection	279

18.1 Preamble

Buffers are not that low level but it makes sense to discuss them in this perspective because it relates to tokenization, internal representation and manipulating.

In due time we can describe some more commands and details here. This is a start. Feel free to tell me what needs to be explained.

18.2 Encoding

Normally processing a document starts with reading from file. In the past we were talking single bytes that were then maps onto a specific input encoding that itself matches the encoding of a font. When you enter an 'a' its (normally ascii) number 97 becomes the index into a font. That same number is also used in the hyphenator which is why font encoding and hyphenation are strongly related. If in an eight bit T_EX engine you need a precomposed 'ä' you have to use an encoding that has that character in some slot with again matching fonts and patterns. The actually used font can have the *shapes* in different slots and remapping is then done in the backend code using

In eight bit environments all this brings a bit of a resource management nightmare along with complex installation of new fonts. It also puts strain on the macro package, especially when you want to mix different input encodings onto different font encodings and thereby pattern encodings in the same document. You can compare this with code pages in operating system, but imagine them potentially being mixed in one document, which can happen when you mix multiple languages where the accumulated number of different characters exceeds 256. You end up switching between

encodings. One way to deal with it is making special characters active and let their meaning differ per situation. That is for instance how in MkII we handled utf8 and thereby got around distributing multiple pattern files per language as we only needed to encoding them in utf and then remap them to the required encoding when loading patterns. A mental exercise is wondering how to support cjk scripts in an eight bit MkII, something that actually can be done with some effort.

The good news is that when we moved from MkII to MkIV we went exclusively utf8 because that is what the LuaTeX engine expects. Upto four bytes are read in and translated in code character. The internal representation is a 32 bit integer (four bytes) instead of a single TeXt) but I'm pretty sure that nowadays no one uses input other than utf8. While ConTeXt is normally quite upward compatible this is one area where there were fundamental changes.

There is still some interpretation going on when reading from file: for instance, we need to normalize the Unicode input, and we feed the engine separate lines on demand. Appendix 18.3 Performance

- *Example code has to be typeset as-is, so braces etc. are just that.* This means that we have to be careful with the placement macros.
- *Content is collected and used later.* A separation of content and usage later on often helps in “including that buffer when a table is placed” using the placement macros.
- *Embedded MetaPost and Lua code.* These languages come with different interpretation of the content.
- *The content comes from a different source.* Examples are xml files where angle brackets are used for tags.
- *The content is generated.* It can for instance come from Lua, where bytes (representing characters) are used for the content.

For these reasons ConTeXt always had ways to store data in ways that makes this possible.

18.3 Performance

When TeX came around, the bottlenecks in running TeX were the processor, memory and disk [2] .. to show up. It was possible to run TeX on a personal computer but it was somewhat pass) trickery was demanding.

When processors became faster and memory plenty the disk became the bottleneck, but that changed when ssd's showed up. Combined with already present file caching that had some impact. We are now in a situation that cpu cores don't get that much faster (at least not twice as fast per iteration) and with TeX being a single core byte cruncher we TeXt code, combined with LuaMetaTeX will give you what you need with a reasonable performance' are made. Inefficient Lua and TeX code has way more impact than storing a few mor

18.4 Files

Nearly always files are read once per run. The content (mixed with commands) is scanned and macros are expanded and/or text is typeset as we go. Internally the LuaMetaTeX engine is in “scanning from file”, “scanning from token lists”, or “scanning from Lua output” mode. The first mode is (in principle) the slowest because utf sequences are converted to tokens (numbers) but there is no way around it. The second method is fast because we already have these numbers, but we need to take into account where the linked list of tokens comes from. If it is converted runtime from for instance file input or macro expansion we need to add the involved overhead. But scanning a stored macro body is pretty efficient especially when the macro is part of the loaded macro package (format file). The third method is comparable with reading from file but here we need to add the overhead involved with storing the Lua output into data structures suitable for TeX's input mechanism, which can involve m MetaTeX we even went a bit further, also because we know what kind of input, processing a

When reading from file or Lua output we interpret bytes turned utf numbers and that is when catcode regimes kick in: characters are interpreted according to the catcode properties: escape character (backslash), curly braces (grouping and arguments), dollars (math), etc. While with reading from token lists these catcodes are already taken care of and we're basically interpreting meanings instead of characters. By changing the catcode regime we can for instance typeset content verbatim from files and Lua strings but when reading from token lists we're sort of frozen. There are tricks to reinterpret the token list but that comes with overhead and limitations.

18.5 Macros

A macro can be seen as a named token with a meaning attached. In LuaMetaTeX macros code character doesn't need all four bytes of an integer and because in the engine numbers

```
\def\foo{abc} \foo \foo \foo
```

When the engine fetches a token from a list it will interpret the command and when it fetches from file it will create tokens on the fly and then interpret those. When a file or list is exhausted the engine pops the stack and continues at the previous level. Because macros are already tokenized they are more efficient than file input. For more about macros you can consult the low level document about them.

The more you use a macro, the more it pays off compared to a file. However don't overestimate this, because in the end the typesetting and expanding all kind of other involved macros might reduce the file overhead to noise.

18.6 Token lists

A token list is like a macro but is part of the variable (register) system. It is just a list (so no arguments) and you can append and prepend to that list.

```
\toks123={abc}      \the\toks123
\scratchtoks{abc} \the\scratchtoks
```

Here `\scratchtoks` is defined with `\newtoks` which creates an efficient reference to a list so that, contrary to the first line, no register number has to be scanned. There are low level manuals about tokens and registers that you can read if you want to know more about this. As with macros the list in this example is three tokens long. Contrary to macros there is no macro overhead as there is no need to check for arguments.²⁵

Because they use more or less the same storage method macros and token list registers perform the same. The power of registers comes from some additional manipulators in `LuaTeX` (and `LuaMetaTeX`) and the fact that one can control expansion with `\the`,

18.7 Buffers

Buffers are something specific for `ConTeXt` and they have always been part of this system. A

```
\startbuffer[one]
line 1
line 2
\stopbuffer
```

Among the operations on buffers the next two are used most often:

```
\typebuffer[one]
\getbuffer[one]
```

Scanning a buffer at the `TeX` end takes a little effort because when we start reading the ca

```
\startluacode
buffers.assign("one",[
line 1
line 2
])
```

²⁵ In `LuaMetaTeX` a macro without arguments is also quite efficient.

`\stopluacode`

Always keep in mind that buffers eventually are read as files: character by character, and at that time the content gets (as with other files) tokenized. A buffer name is optional. You can nest buffers, with and without names.

Because `ConTEXt` is very much about re-use of content and selective processing we have an

The `\definebuffer` command defines a new buffer environment. When you set buffers in Lua you don't need to define a buffer because likely you don't need the `\start` and `\stop` commands. Instead of `\getbuffer` you can also use `\getdefinedbuffer` with defined buffers. In that case the `before` and `after` keys of that specific instance are used.

The `\getinlinebuffer` command, which like the getters takes a list of buffer names, ignores leading and trailing spaces. When multiple buffers are flushed this way, spacing between buffers is retained.

The most important aspect of buffers is that the content is *not* interpreted and tokenized: the

```
\definebuffer[MyBuffer]
```

```
\startMyBuffer
```

```
\bold{this is  
a buffer}
```

```
\stopMyBuffer
```

```
\typeMyBuffer \getMyBuffer
```

These commands result in:

```
\bold{this is  
a buffer}
```

this is a buffer

There are not that many parameters that can be set: `before`, `after` and `strip` (when set to no leading and trailing spacing will be kept). The `\stop...` command, in our example `\stopMyBuffer`, can be defined independent to so something after the buffer has been read and stored but by default nothing is done.

You can test if a buffer exists with `\doifelsebuffer` (expandable) and `\doifelsebufferempty` (unexpandable). A buffer is kept in memory unless it gets wiped clean with `resetbuffer`.

```
\savebuffer      [MyBuffer][temp]      % gets name: jobname-temp.tmp
\savebufferinfile[MyBuffer][temp.log] % gets name: temp.log
```

You can also stepwise fill such a buffer:

```
\definesavebuffer[slide]
```

```
\startslide
  \starttext
\stopslide
\startslide
  slide 1
\stopslide
text 1 \par
\startslide
  slide 2
\stopslide
text 2 \par
\startslide
  \stoptext
\stopslide
```

After this you will have a file `\jobname-slide.tex` that has the two lines wrapped as text. You can set up a ‘save buffer’ to use a different filename (with the `file` key), a different prefix using `prefix` and you can set up a directory. A different name is set with the `list` key.

You can assign content to a buffer with a somewhat clumsy interface where we use the delimiter `\endbuffer`. The only restriction is that this delimiter cannot be part of the content:

```
\setbuffer[name]here comes some text\endbuffer
```

For more details and obscure commands that are used in other commands you can peek into the source.

Using buffers in the `cld` interface is tricky because of the `catcode` magick that is involved but there are setters and getters:

function	arguments
<code>buffers.assign</code>	name, content [,catcodes]
<code>buffers.erase</code>	name

```

buffers.prepend      name, content
buffers.append       name, content
buffers.exists       name
buffers.empty        name
buffers.getcontent   name
buffers.getlines     name

```

There are a few more helpers that are used in other (low level) commands. Their functionality might adapt to their usage there. The `context.startbuffer` and `context.stopbuffer` are somewhat differently defined than regular `cld` commands.

18.8 Setups

A setup is basically a macro but is stored and accessed in a namespace separated from ordinary macros. One important characteristic is that inside setups newlines are ignored.

```

\startsetups MySetupA
  This is line 1
  and this is line 2
\stopsetups

```

```
\setup{MySetupA}
```

This is line 1 and this is line 2

A simple way out is to add a comment character preceded by a space. Instead you can also use `\space`:

```

\startsetups [MySetupB]
  This is line 1 %
  and this is line 2\space
  while here we have line 3
\stopsetups

```

```
\setup[MySetupB]
```

This is line 1 and this is line 2 while here we have line 3

You can use square brackets instead of space delimited names in definitions and also in calling up a (list of) setup(s). The `\directsetup` command takes a single setup name and is therefore more efficient.

Setups are basically simple macros although there is some magic involved that comes from their usage in for instance xml where we pass an argument. That means we can do the following:

```
\startsetups MySetupC
  before#1after
\stopsetups
```

```
\setupwithargument{MySetupC}{ {\em and} }
```

before_and_after__

Because a setup is a macro, the body is a linked list of tokens where each token takes 8 bytes of memory, so MySetupC has 12 tokens that take 96 bytes of memory (plus some overhead related to macro management).

18.9 xml

Discussing xml is outside the scope of this document but it is worth mentioning that once an xml tree is read is, the content is stored in strings and can be filtered into T_EX, where it is interpreted as if coming from files (in this case Lua strings). If needed the

18.10 Lua

As mentioned already, output from Lua is stored and when a Lua call finishes it ends up on the so called input stack. Every time the engine needs a token it will fetch from the input stack and the top of the stack can represent a file, token list or Lua output. Interpreting bytes from files or Lua strings results in tokens. As a side note: Lua output can also be already tokenized, because we can actually write tokens and nodes from Lua, but that's more an implementation detail that makes the Lua input stack entries a bit more complex. It is normally not something users will do when they use Lua in their documents.

18.11 Protection

When you define macros there is the danger of overloading some defined by the system. Best use CamelCase so that you stay away from clashes. You can enable some checking:

```
\enabledirectives[overloadmode=warning]
```

or_when_you_want_to_quit_on_a_clash:_

```
\enabledirectives[overloadmode=error]
```

When_these_trackers_are_enabled_you_can_get_around_the_check_with:_

```
\pushoverloadmode
```

```
...
```

```
\popoverloadmode
```

But_delay_that_till_you're_sure_that_redefining_is_okay._

18.11 Colofon

Author	Hans Hagen
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com contextgarden.net ntg-context@ntg.nl

19 Accuracy

low level

TEX

accuracy

Contents

19.1	Introduction	282
19.2	Posits	283
19.3	MetaPost	286
19.4	Lua	287

19.1 Introduction

This is work in progress, uncorrected.

When you look at T_EX and MetaPost output the accuracy of the rendering stands out, unless you use portable) floating point calculations, it does all with 32 bit integers, except in the backend where glue calculations are used for finalizing the glue values. It all changed a bit when we added Lua because there we mix integers and doubles but in practice it works out okay.

When looking at floating point (and posits) one can end up in discussions about which one is better, what the flaws of each are, etc. Here we're only interested in the fact that posits are more accurate in the ranges where T_EX and MetaPost operate, as well as the

When you work with dimensions like points, they get converted to an integer number (the sp unit) and from that it's just integer calculations. The maximum dimension is 16383.99998pt, which already shows a rounding issue. Of course when one goes precise for sure there is some loss, but on the average we're okay. So, in the next example the two last rows are equivalent:

```
.1pt 0.1pt 6554sp
.2pt 0.2pt 13107sp
.3pt 0.3pt 19661sp
.1pt + .2pt 0.3pt 19661sp
```

When we're at the Lua end things are different, there numbers are mapped onto 64 bit floating point variables (doubles) and not all numbers map well. This is what we get when we work with doubles in Lua:

```
.1          0.1
.2          0.2
.3          0.3
.1 + .2 0.30000000000000004
```

The serialization looks as if all is okay but when we test for equality there is a problem:

```
.3 == .3 true
.1 + .2 == .3 false
```

This means that a test like this can give false positives or negatives unless one tests the difference against the accuracy (in MetaPost we have the `eps` variable for that). In \TeX clipping of the decimal fraction influences equality.

```
\iflua{ .3 == .3 } Y\else N\fi different
\iflua{ .1 + .2 == .3 } Y\else N\fi different
```

The serialization above misguides us because the number of digits displayed is limited. Actually, when we would compare serialized strings the equality holds, definitely within the accuracy of \TeX . But here is reality:

	.3	.1 + .2
%0.10g	0.3	0.3
%0.17g	0.29999999999999999	0.30000000000000004
%0.20g	0.2999999999999999889	0.30000000000000004441
%0.25g	0.2999999999999999888977698	0.300000000000000044408921

The above examples use 0.1, 0.2 and 0.3 and on a 32 bit float that actually works out okay, but LuaMeta \TeX is 64 bit. Is this really important in practice? There are indeed ca

19.2 Posits

The next table shows the same as what we started with but with a different serialization.

```
.1          0.1
.2          0.2
.3 0.30000001
.1 + .2 0.30000001
```

And here we get equality in both cases:

```
.3 == .3 true
.1 + .2 == .3 true
```

The next table shows what we actually are dealing with. The `\if`-test is not a primitive but provided by Con \TeX t.

```
\ifpositunum{ .3 == .3 } Y\else N\fi equal
\ifpositunum{ .1 + .2 == .3 } Y\else N\fi equal
```

And what happens when we do more complex calculations:

```
math .sin(0.1 + 0.2) == math .sin(0.3) false
posit.sin(0.1 + 0.2) == posit.sin(0.3) true
```

Of course other numbers might work out differently! I just took the simple tests that came to mind.

So what are these posits? Here it's enough to know that they are a different way to store numbers with fractions. They still can loose precision but a bit less on smaller values and often we have relative small values in \TeX . Here are some links:

<https://www.johngustafson.net/pdfs/BeatingFloatingPoint.pdf>
<https://posithub.org/conga/2019/docs/14/1130-FlorentDeDinechin.pdf>

There are better explanations out there than I can provide (if at all). When I first read about these unums (a review of the 2015 book “The End of Error Unum Computing”) I was intrigued and when in 2023 I read something about it in relation to RISCV I decided to just add this playground for the users. After all we also have decimal support. And interval based solutions might actually be good for MetaPost, so that is why we have it as extra number model. There we need to keep in mind that MetaPost in non scaled models also apply some of the range checking and clipping that happens in scaled (these magick 4096 tricks).

For now it is enough to know that it's an alternative for floats that *could* work better in some cases.

But how about \TeX ? Per April 2023 the LuaMeta \TeX engine has native support for floats (through \TeX t). How that works can be demonstrated with some examples. The float related commands are:

```
\scratchdimen=1.23456pt
\scratchfloat=1.23456
```

We now use these two variables in an example:

```
\setbox0\hbox to \scratchdimen {x}\the\wd0
\scratchdimen \dimexpr \scratchdimen * 2\relax
\setbox0\hbox to \scratchdimen {x}\the\wd0
```

²⁶ Are 64 bit posits actually being worked on in softposit? There are some commented sections. We also need to patch some unions to make it compile as C.

```
\advance \scratchdimen \scratchdimen
\setbox0\hbox to \scratchdimen {x}\the\wd0
\multiply\scratchdimen by 2
\setbox0\hbox to \scratchdimen {x}\the\wd0
```

1.23456pt
2.46912pt
4.93823pt
9.87646pt

When we use floats we get this:

```
\setbox0\hbox to \scratchfloat {x}\the\wd0
\scratchfloat \floatexpr \scratchfloat * 2\relax
\setbox0\hbox to \scratchfloat {x}\the\wd0
\advance \scratchfloat \scratchfloat
\setbox0\hbox to \scratchfloat {x}\the\wd0
\multiply\scratchfloat by 2
\setbox0\hbox to \scratchfloat {x}\the\wd0
```

1.23456pt
2.46912pt
4.93823pt
9.87648pt

So which approach is more accurate? At first sight you might think that the dimensions are better because in the last two lines they indeed duplicate. However, the next example shows that with dimensions we lost some between steps.

```
\scratchfloat \floatexpr \scratchfloat * 2\relax \the\scratchfloat
\advance \scratchfloat \scratchfloat \the\scratchfloat
\multiply\scratchfloat by 2 \the\scratchfloat
```

1.2345599979162216187
2.4691199958324432373
4.9382399916648864746
9.8764799833297729492

One problem with accuracy is that it can build up. So when one eventually does some comparison the expectations can be wrong.

```
\dimen0=1.2345pt
```

```
\dimen2=1.2345pt
```

```
\ifdim          \dimen0=\dimen2 S\else D\fi \space +0sp: [dim]
\ifinterval\dim0sp\dimen0 \dimen2 0\else D\fi \space +0sp: [0sp]
```

```
\advance\dimen2 1sp
```

```
\ifdim          \dimen0=\dimen2 S\else D\fi \space +1sp: [dim]
\ifinterval\dim 1sp \dimen0 \dimen2 0\else D\fi \space +1sp: [1sp]
\ifinterval\dim 1sp \dimen2 \dimen0 0\else D\fi \space +1sp: [1sp]
\ifinterval\dim 2sp \dimen0 \dimen2 0\else D\fi \space +1sp: [2sp]
\ifinterval\dim 2sp \dimen2 \dimen0 0\else D\fi \space +1sp: [2sp]
```

```
\advance\dimen2 1sp
```

```
\ifinterval\dim 1sp \dimen0\dimen2 0\else D\fi \space +2sp: [1sp]
\ifinterval\dim 1sp \dimen2\dimen0 0\else D\fi \space +2sp: [1sp]
\ifinterval\dim 5sp \dimen0\dimen2 0\else D\fi \space +2sp: [5sp]
\ifinterval\dim 5sp \dimen2\dimen0 0\else D\fi \space +2sp: [5sp]
```

Here we show a test for overlap in values, the same can be done with integer numbers (counts) and floats. This interval checking is an experiment and we'll see it if gets used.

```
S_+0sp:_[dim]_0_+0sp:_[0sp]_
D_+1sp:_[dim]_0_+1sp:_[1sp]_0_+1sp:_[1sp]_0_+1sp:_[2sp]_0_+1sp:_[2sp]_
D_+2sp:_[1sp]_D_+2sp:_[1sp]_0_+2sp:_[5sp]_0_+2sp:_[5sp]_
```

There are also `\ifintervalfloat` and `\ifintervalnum`. Because I have worked around these few scaled point rounding issues for decades, it might actually take some time before we see the interval tests being used in `ConTEXt`. After all, there is no reason to touch

To come back to posits, just to be clear, we use 32 bit posits and not 32 bit floats, which we could have but that way we gain some accuracy because less bits are used by default for the exponential.

In `ConTEXt` we also provide a bunch of pseudo primitives. These take one float: `\pfsin`, `\pf`

19.3 MetaPost

In addition to the instances `metafun` (double in `LMTX`), `scaledfun`, `doublefun`, `decimalfun` we now also have `positfun`. Because we currently use 32 bit posits in the new number system there is no real gain over the already present 64 bit doubles. When 64 bit posits show up we might move on to that.

19.4 Lua

We support posits in Lua too. Here we need to create a posit user data object. The usual metatable magick kicks in:

```
local p = posit.new(123.456)
local q = posit.new(789.123)
local r = p + q
```

Here we just mention what is currently interface. The management functions are: `new`, `copy`, `number`, `integer`, `rounded`, `toposit` and `fromposit`. The usual operators are also supported: `+`, `-`, `*`, `/`, `^`, as well as the binary `|`, `&`, `~`, `<<` and `>>`. We can compare with `==`, `>=`, `more_verbose`, `bor`, `bxor`, `band`, `shift`, `rotate` are there too.

There is a subset of math provided: `min`, `max`, `abs`, `conj`, `modf`, `acos`, `asin`, `atan`, `ceil`, `cos`, what special are `NaN` and `NaR`.

Currently integer division (`//`) and modulo (`%`) are not available, but that might happen at some time.

19.4 Colofon

Author	Hans Hagen
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com contextgarden.net ntg-context@ntg.nl

20 Balancing



low level

TEX

balancing

Contents

20.1	Introduction	289
20.2	Intercepting the MVL	289
20.3	Balancing	292
20.4	Forcing breaks	297
20.5	Marks	298
20.6	Inserts	298
20.7	Discardables	302
20.8	Passes	304
20.9	Passes	305

20.1 Introduction

This is work in progress as per end 2024 these mechanisms are still in flux. We expect them to be stable around the ConT_EXt meeting in 2025. The text is not corrected, so f

This manual is about a new (sort of fundamental) feature that got added to LuaMeta-_T_E_X when we started upgrading column sets. In _T_E_X we have a par builder that does a multi-pass optimization where it considers various solutions based on tolerance, penalties, demerits etc. The page builder on the other hand is forward looking and backtracks to a previous break when there is an overflow. The balancing mechanism discussed here is basically a page builder operating like the par builder: it looks at the whole picture.

In order to make this a useful mechanism the engine also permits intercepting the main vertical list, so we start by introducing this.

20.2 Intercepting the MVL

When content gets processed it's added to a list. We can be in horizontal mode or vertical mode (let's forget about math mode). In vertical mode we can be in a box context (say `\vbox`) or in what is called the main vertical list: the one that makes the page. But what is page? When _T_E_X has collected enough to match the criteria set by `\p`

For various mechanisms it matters if they are used inside a contained boxed environment or in the more liberal main vertical list (from now on called mvl). That's why we can intercept the mvl and use it later. Intercepting works as follows:

```
\beginmvl 1
```

```
various content
\endmvl
```

```
\beginmvl 2
various content
\endmvl
```

When_at_some_point_you_want_this_content,_you_can_do_this:_

```
\setbox\scratchboxone\flushmvl 2
\setbox\scratchboxtwo\flushmvl 1
```

and_then_do_whatever_is_needed._You_can_see_what_goes_on_with:_

```
\tracingmvl 1
```

There_is_not_much_more_to_say_other_than_that_this_is_the_way_to_operate_on_content_ as_if_it_were_added_to_the_page_which_can_be_different_from_collecting_something_in_ a_vertical_box._Think_of_various_callbacks_that_can_differ_for_the_mvl_and_a_box._

The_\beginmvl_primitive_takes_a_number_or_a_set_of_keywords,_as_in:_

```
\beginmvl
  index    1
  options \numexpr "01 + "04\relax
\relax
```

There_is_of_course_some_possible_interference_with_mechanism_that_check_the_page_ properties_like_\pagegoal._If_needed_one_can_check_this:_

```
\ifcase\mvlcurrentlyactive
  % main mvl
\or
  % first one
\else
  % other ones
\fi
```

Possible_applications_of_this_mechanism_are_the_mentioned_columns_and_parallel,_in- dependent,_streams._However_for_that_we_need_to_be_able_to_manipulate_the_collected_ content._Actually,_the_next_manipulator_preceded_the_capturing,_because_we_first_wanted_ to_make_sure_that_what_we_had_in_mind_made_sense._


The `\beginmvl` also accepts keywords. You can specify an index (an integer), a prevdepth (dimensions) and options (an integer bitset). Possible option bit related values are:

```
0x1 ignore prevdepth \ignoreprevdepthmvloptioncode
0x2 no prevdepth     \noprevdepthmvloptioncode
0x4 discard top      \discardtopmvloptioncode
0x8 discard bottom   \discardbottommvloptioncode
```

Here the last column is a numeric alias available in ConTeXt. More options are likely to show


```
\beginmvl
  index      1
  prevdepth  0pt
  options    \discardtopmvloptioncode
\relax
\scratchdimen\prevdepth
\dontleavehmode
\quad\the\mvlcurrentlyactive\quad\the\scratchdimen
\quad\blackrule[height=\strutht,depth=\strutdp,color=darkred]
\endmvl
```

```
\ruledhbox {\llap{1\quad}}\flushmvl 1}
```

1 

```
\beginmvl
  index  2
  options \numexpr
          \ignoreprevdepthmvloptioncode
          + \discardtopmvloptioncode
\relax
\relax
\scratchdimen\prevdepth
\dontleavehmode
\quad\the\mvlcurrentlyactive\quad\the\scratchdimen
\quad\blackrule[height=\strutht,depth=\strutdp,color=darkred]
\endmvl
```

```
\ruledhbox {\llap{2\quad}}\flushmvl 2}
```

2 

```
\beginmvl 3 % when no keywords are used we expect a number
\scratchdimen\prevdepth
```

```

\dontleavehmode
\quad\the\mvlcurrentlyactive\quad\the\scratchdimen
\quad\blackrule[height=\strutht,depth=\strutdp,color=darkred]
\endmvl

```

```
\ruledhbox {\llap{3\quad}\flushmvl 3}
```

3

```
3 0.0pt
```

```

\beginmvl index 4 options 1
\scratchdimen\prevdepth
\dontleavehmode
\quad\the\mvlcurrentlyactive\quad\the\scratchdimen
\quad\blackrule[height=\strutht,depth=\strutdp,color=darkred]
\endmvl

```

```
\ruledhbox {\llap{4\quad}\flushmvl 4}
```

4

```
4 -1000.0pt
```

20.3 Balancing

Balancing is not referring to balancing columns but to ‘a result that looks well balanced’. Just like we want lines in a paragraph to look consistent with each other, something that is reflected in the (adjacent) demerits, we want the same with vertical split of pieces. For this purpose we took elements of the `par` builders to construct a (page) snippet builder. Here are some highlights:

- Instead of a pretolerance, tolerance and emergency pass we only enable the last two. In the `par` builder the pretolerance pass is the one without hyphenation.
- We seriously considered vertical discretionaries but eventually rejected the idea: we just don't expect users to go through the trouble of adding lots of split related `pre`, `post` and `replace` content. It's not hard to support it but in the end it also interfered with other demands that we had. We kept the code around for a while but then removed it. To mention one complication: if we add some new node we also need to intercept it in various callbacks that we already have in place in `ConTeXt`. As with horizontal discretionaries, we then need to go into the components and some
- As with the `par` builder, `TeX` will happily produce an overfull box when no solution is pos

- In many cases there is no stretch available. There are also widow, club, shape and orphan penalties that can limit the solution space.
- When we look at splitting pages (and boxes) we see (split) top skip kick in. This is something that we need to provide one way or the other. And as we have to do that, we can as well provide support for bottom skip. A horizontal analogue is protrusion, something that also has to be taken into account in a rather dynamic way, at the beginning or end of the currently analyzed line.
- There is no equivalent of hanging indentation but a shape makes sense. Here the shape defines heights, top and bottom skips and maybe more in the future. For that reason we use a keyword driven shape.
- Because we have so called par passes, it made sense to have something similar for balancing. This gives us the opportunity to experiment with various variables that drive the process.
- For those who read what we wrote about the par builder, it will not come as surprise that we also added extensive tracing and a callback for intercepting the results. This makes it possible to show the same detailed output as we can do for par passes.

It's about time for some examples but before we come to that it is good to roughly explain how the page builder works. When the page builder is triggered it will take elements from the contributions list and add them to the page. When doing that it keeps track of the height and depth as contributed by boxes and rules. Because it will discard glue and kerns it does some checking there. An important feature is that the depth is added in a next iteration. The routine also needs to look at inserts. The variables `\pagegoal` (original `\vsize` minus accumulated insert heights) and `\pagetotal` are compared and when we run over the target height the accumulated stretch and shrink in glue (when present) will be used to determine how bad this break is. If it is too bad, the previous best break will be taken. Penalties can make a possible break more or less attractive. When the output routine gets a split of page, the total is not reliable because we can have backtracked to the previous break. In LuaMetaTeX we have some

In order to make the first lines align properly relative to the top of the page there is a variable `\topskip`. The height of the first line is at least that amount. The correction is calculated when the first contribution happens: a box or rule.

When we look at the balancer it is good to keep in mind that where the page builder stepwise adds and checks, the balancer looks at the whole picture. The page builder does a decent job but is less sophisticated than the par builder. There is a badness

calculation, penalties are looked at, glue is taken into account but there are no demerits.

We want the balancer to work well with column sets that are very much grid based. But in getting there we had some hurdles to take. Because the algorithm (like the par builder) happily results in overfull boxes unless emergency stretch is set, pages can overflow. When there is no stretch and/or shrink using emergency stretch can give an underfull page.

The way out of this is to have non destructive trial passes and decrease the number of lines. Of course we can get short pages but when for instance it concerns a section title that gets moved this is no big deal. In a similar fashion splitting a multi-line formula is also okay.

- Collect the content in an mvl list and after that's done put the result in a box.
- Set up a balance shape that specifies the slots in columns (normally a column is just a blob of text).
- Perform a trial balance run. As soon as an overfull page is seen, adapt the balance shape and do a new trial run.
- When we're fine, either because we reached the end without overfull column or by passing the set deadcycles value, quit the trial process and balance the original list using the most recent balance shape.
- Flush the result by fetching the topmost from the result split collection and feed it into the page flow. The boxed pseudo page will happily trigger the output routine that in turn constructs the final page.

At some point we decided to support multiple mvl streams and therefore changed the last mentioned step. Because we store the whole column set we can as well also store the assembled page bodies. This way we can flush different streams into the same result.

- Flush the result by fetching the topmost from the result split collection and feed it into the page flow. Do this for every saved (mvl) stream.
- When we're done, the boxed pseudo pages will be flushed as pages. In the process, for every page we identify marks.

We are now ready to look at some examples. Here we also show what balance shapes do. These basically describe a sequence of slots to be filled. The last specification is used when we exceed the number of defined slots. These are just examples of simple situations, for real applications more code is needed.

Balancing

We start with some content in a box. This can of course be a flushed mvl but here we just set it directly:

```
\setbox\scratchboxone\vbox\bgroup
  \hsize.30\hsize
  \samplefile{tufte}
\egroup
```

We will split this box in columns. If you are familiar with \TeX you might know that a paragraph

```
\balanceshape 3
  vsize      12\lineheight
  topskip    \strutht
  bottomskip \strutdp
next
  vsize      5\lineheight
  topskip    \strutht
  bottomskip \strutdp
next
  vsize      8\lineheight
  topskip    \strutht
  bottomskip \strutdp
\relax
```

```
\setbox\scratchboxtwo\balance\scratchboxone
```

Contrary to a `\parshape`, a `\balanceshape` is not wiped after the work is done. It also expects keys and values. As with `\par` passes each step is separated by `next`. This makes it an extensible mechanism. Finally we will split the box according to this shape:

```
\hbox \bgroup
  \localcontrolledendless {%
    \ifvoid\scratchboxtwo
      \expandafter\quitloop
    \else
      \setbox\scratchbox\ruledhbox\bgroup
        \vbalancedbox\scratchboxtwo
      \egroup
      \vbox to 12\lineheight \bgroup
        \box\scratchbox
        \vfill
      \egroup
    }
```

```

\hskip1em
\fi
}\unskip
\egroup

```

The_result_is_shown_here:

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

Like the par_builder we can end up with overfull boxes but we can deal with that by using trial runs.

```
\setbox\scratchboxtwo\vbalance\scratchboxone trial
```

In that case the result is made from empty boxes so the original is not disturbed. Here we show an overflow, so in the first resulting box you can compare the height with the requested one and when it's larger you can decide to decrease the first height in the shape and try again.

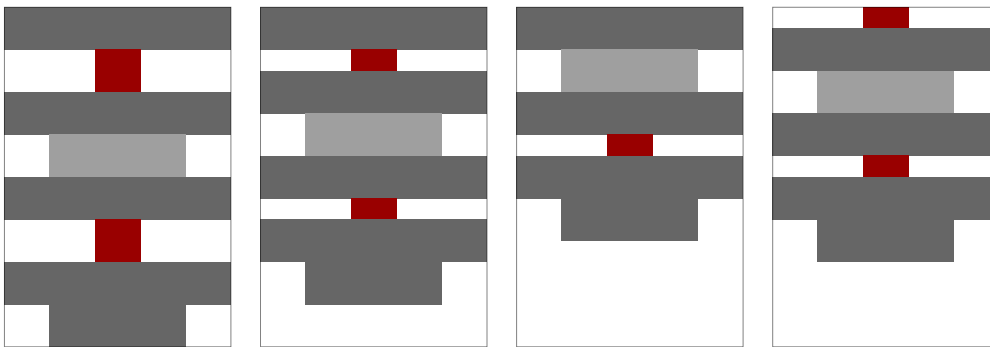
Many readers will skim over formulas on their first reading of your exposition. Therefore, your sentences should flow smoothly when all but the simplest formulas are replaced by “blah” or some other grunting noise.	Many readers will skim over formulas on their first reading of your exposition. Therefore, your sentences should flow smoothly when all but	the simplest formulas are replaced by “blah” or some other grunting noise.
<div style="border: 1px solid black; padding: 2px; width: fit-content;">test</div>		

Of course that involves some juggling of the shape but after all we have Lua at our disposal so in the end it's all quite doable.

	real	target
1	169.89122pt	156.95874pt
2	65.39948pt	65.39948pt
3	51.17216pt	104.63916pt

Because the balancer can produce what otherwise the page builder produces, we need to handle the equivalent of `top_skip` which is what the already shown `top_keyword` takes care of. This means that the current slice (think current line in the `par_builder`) has to take that into account. This can be compared to the left- and right protrusion in the `par_builder`. When we typeset on a grid we have an additional demand.

When we surround (for instance a formula) with `halfline` spacing, we eventually have to return on the grid. One complication is that when we are in grid mode and use `half_line` vertical spacing, we can end up in a situation where the initial half line space is on a previous page. That means that we need to use a larger `top_skip`. This is not something that we want to burden the balancer with but we have ways to trick it into taking that compensation into account.



However, when we split in the middle of that segment, we can end up with a half line skip in a next slot because \TeX will remove glue at the edge. So we end up with what

20.4 Forcing breaks

Because the initial application of balancing was in column sets, we also need the ability to goto a next slot (step in a shape), column (possibly more steps), page (depending on the page state), and spread (for instance if we are doubles ided). For this we use `\balanceboundary`. It takes two values and when the boundary node triggers a callback in the builder these are passed along with a shape identifier and current shape slot. That callback can then signal back that we need to try a break here with a given penalty. Assuming that at the Lua end we know at which slot we have a slot, column, page or spread break. Multiple slots can be skipped by multiple boundaries. There is one pitfall: we need something in a slot in order to break at all, so one ends up with for instance:

Forcing breaks

```

\balanceboundary 3 1\relax
\vskip\zeropoint
\balanceboundary 3 0\relax
\vskip\zeropoint
\balanceboundary 3 0\relax

```

Here the 3 is just some value that the callback can use to determine its action (like goto a next page) and the second value provides a detail. Of course all depends on the intended usage. By using a callback we can force breaks while not burdening the engine with some hard coded solution. For example, in ConT_EXt we used these (the value

first	second	action	user interface
1	1 or 0	goto next spread (1 initial, 0 follow up)	\page[spread]
2	1 or 0	goto next page (idem)	\page
3	1 or 0	goto next column (idem)	\column
4	1 or 0	goto next slot (idem)	\column[slot]
5	n	next slot when more than n lines	\testroom[5]
6	s	next slot when more than s scaled points	\testroom[80pt]

20.5 Marks

It is possible to synchronize the marks with those in the results of balanced segments with a few Lua helpers that do the same as the page builder does at the start of a page, while packaging the page and when wrapping it up. So, instead of split marks we can have real marks.

20.6 Inserts

Before we go into detail, we want to point out that when implementing a (balancing) mechanism as introduced above, decisions have to be made. In traditional T_EX there is for instance T_EX and LuaMetaT_EX it's (ConT_EXt) user demands and challenges that drives what gets impl

Users on the other hand have come up with demands for columns, typesetting on the grid, multiple notes, balancing, and parallel content streams. The picture we get from that makes us confident that what we provide is generally enough and as users understand the issues at hand (maybe as side effect of struggling with solutions) it's not that hard to explain why constraints are in place. It makes more sense to have a limited reliable mechanism that deals with the kind of (foot)notes that known users need than to cook up some complex mechanism that caters potential specific demands by potential users. Of course we have our own challenges to deal with, even if the resulting features will probably not be used that often. So here are the criteria that make sense:

- We can assume a reasonable amount of notes.
- These are normally small with no (vertical) whitespace.
- Notes taking multiple lines may split.
- But we need to obey widow and club penalties.
- There can be math formulas but mostly inline.
- We need to keep them close to where they are referred from.

But,

- We can ignore complex conflicting demands.
- As long as we get some result, we're fine.
- So users have to check what comes out.
- We don't assume fully automated unattended usage.

And of course:

- Performance should be acceptable.
- User interfaces should be intuitive.
- Memory consumption should be reasonable.

We have users who use multiple note classes so that also has to be handled but again we don't need to come up with solutions that solve all possible demands. We can assume that when a book is published that needs them, the author will operate within the constraints.

We mentioned footnotes being handled by the page builder so how about them in these balanced slots? Given the above remarks, we assume sane usage, so for instance columns that have a single slot with possibly fixed content at the top or bottom (and maybe as part of the stream). The balancer handles notes by taking their height into account and when a result is used one can request the embedded inserts and deal with them. Again this is very macro package dependent. Among the features dealt with are space above and between a set of notes, which means that we need to identify the first and successive notes in a class. Given how the routine works, this is a dynamic feature of a line: the amount of space needed depends on how many inserts are within a slot. When we did some extreme tests with several classes of notes and multiple per column we saw runtime increasing because instead of a few passes we got a few hundred. In an extreme case of 800 passes to balance the result we noticed over four million checks for note related spacing. We could bring that down to one tenth so in the end we are still slower but less noticeable. Here are the helper primitives for inserts:

```
<state> = \boxinserts <box>
```

```
<box>    = \vbalancedinsert <box> <class>
<state> = \boxinserts <box>
```

A (foot)note implementation is very macro package dependent so the next example is just that: an example of using the available primitive. We start by populating a mvl with a sample text and a single footnote.

```
\begingroup
  \forgetall
  \beginmvl
    index 5
    options \numexpr
      \ignoreprevdepthmvloptioncode
      + \discardtopmvloptioncode
    \relax
  \relax
  \hsize .4tw
  Line 1 \par Line 2 \footnote {Note 1} \par Line 3 \par
  Line 4 \footnote {Note 2} \par Line 5 \par Line 6 \par
  \endmvl
\endgroup
```

We fetch the footnote number, which is one of many possible defined inserts

```
\cdef\currentnote{footnote}%
\scratchcounter\currentnoteinsertionnumber
```

The quick and dirty balancer uses a simple shape of 5 lines with normal strut properties. From the balanced result we take two columns. We test if there is an insert and take action when there is. Here we just filter the footnotes but there can of course be more. We overlay these notes over (under) the column that has them. So we work per column.

```
\begingroup
  \setbox\scratchboxone\flushmvl 5
  \balanceshape 1
    vsize      5lh
    topskip    1sh
    bottomskip 1sd
  \relax
  \setbox\scratchboxtwo\vbalance\scratchboxone
  \ruledhbox \bgroup
```

```

\localcontrolledrepeat 2 {
  \ifnum\currentloopiterator > 1
    \hskip2\emwidth
  \fi
  \setbox\scratchboxthree\vbalancedbox\scratchboxtwo \relax
  \ifnum\boxinserts\scratchboxthree > 3
    \setbox\scratchboxfour\vbalancedinsert
      \scratchboxthree\scratchcounter
    \wd\scratchboxfour 0pt
    \box\scratchboxfour
  \fi
  \box\scratchboxthree
}\unskip
\egroup
\endgroup

```

The_result_is:_

Line 1	Line 4 ²⁸
Line 2 ²⁷	Line 5
Line 3	Line 6
²⁷ Note 1	²⁸ Note 2

As we progressed we realized that the ‘balancer’ used in column sets can also be used for single columns and we can even support a mix of single and multi columns. There is however a problem: within a mvl we can deal with spacing but we can't do that reliable across mvl's and especially when we cross a page it becomes hard to identify if some (vertical) spacing is needed; we don't want it at the bottom or top of a page. This feature is too experimental to be discussed right now.

We assumed reasonable notes to be used but even if a user tries to keep notes small and avoid too many, there are cases where they might look like a paragraph and when there are more in a row, it might be that a column overflows. This is why we have some support for split notes. This is accomplished by two additional commands:

```

\setbox\scratchboxone\vbalance\scratchboxone\relax
\vbalanceddeinsert\scratchboxone\relax

```

Here we convert inserts in such a way that they are taken into account by the balancer so that multi-slot optimization takes place. Afterwards, when we loop over the result we can reconstruct the inserts:

Inserts

```
\setbox\scratchboxtwo\vbalancedbox\scratchboxone
\vbalancedreinsert\scratchboxtwo\relax
```

Among the reasons that these are explicit actions, is that we want to experiment but also be able to see the effect by selectively enabling it. You can get better results by forcing depth correction.

```
\setbox\scratchboxone\vbalance\scratchboxone
\vbalanceddeinsert\scratchboxone forcedepth\relax
```

This will use the depth as defined by `\insertlinedepth` which is an insert class specific parameter, but discussing details of inserts is not what we do here. The reason for using a `\relax` in the above examples is that we want to stress that when keywords are involved, you need to prevent look-ahead, especially when an `\if...` or expandable loop follows, which is not uncommon when we balance.

It is possible to define top and bottom inserts but of course these need to be filtered and placed at the `TEX` end, so this is macro package specific. Here we just mention that it is

20.7 Discardables

This is a preliminary explanation.

```
\begingroup
  \beginmvl
    index 5
    options \numexpr
      \ignoreprevdepthmvloptioncode
      + \discardtopmvloptioncode
    \relax
  \relax
  \hsize .4tw
  \par
  \vskip0pt
  {\darkred \hrule discardable height 1sh depth 1sd width 1em}
  \par
  % we need the strut because the rule obscures it .. todo
  \dorecurse{8}{\strut Line #1 \par}
  \vskip\zeropoint
  {\darkblue \hrule discardable height 1sh depth 1sd width 1em}
  \par
```


```

\endmvl
\endgroup

\setbox\scratchboxone\flushmvl 5
\balanceshape 1
  vsize      5lh
  topskip    1sh % see comment above
  bottomskip 1sd
  options    3
\relax
\setbox\scratchboxtwo\balance\scratchboxone\relax % lookahead

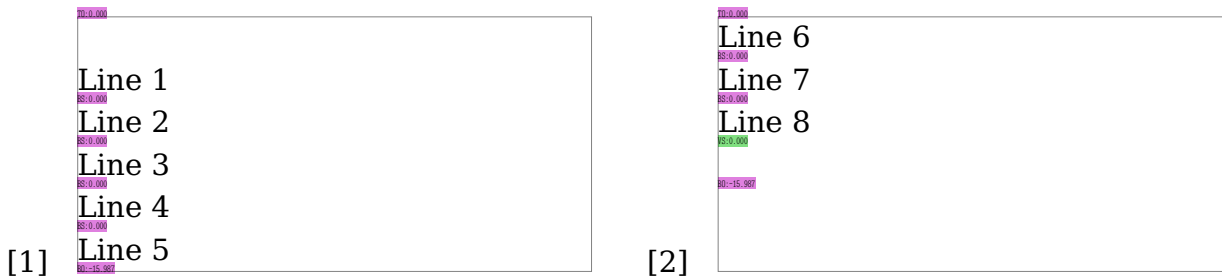
\hpack \bgroup
  \localcontrolledrepeat 3 {
    \ifvoid\scratchboxtwo\else
      \setbox\scratchboxthree\vballedbox\scratchboxtwo
      \ifvoid\scratchboxthree\else
        \dontleavehmode\llap{[\the\currentloopiterator]\quad}%
        \ruledhpack{\box\scratchboxthree}\par
      \fi
      \hskip 4em
    \fi
  }\unskip
\egroup

```

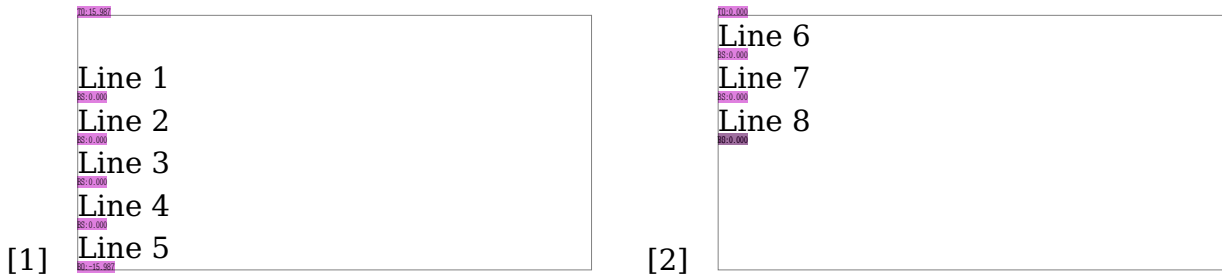
[1] 

[2] 

When at the top, the rule will be ignored and basically sticks out. When at the bottom the rule might end up in a zero dimension box. With `\vballeddiscard\scratchboxtwo` they will become an `\nohrule`. Basically we're talking of optional content. The option set in the shape definition tells if we have a top (1) and/or bottom (2), here we have both.



Here we actually still have the rule but marked as invisible. So, topskip has a negative amount. In the next case the remove keyword makes the rule go away in which case we also adapt the topskip accordingly.



You need to juggle a bit with skips and penalties to get this working as you like. Instead of rules you can also use boxes, for example before:

```
\vskip\zeropoint
\rule\vbox discardable {\hpack{\strut BEFORE}}
\par
```

and after:

```
\forgetall \par \vskip\zeropoint
\rule\vbox discardable {\hpack{\strut AFTER}}%
\penalty\minusone % !
\par
```

It currently is a playground so it might (and probably will) evolve. Although it was also made for a specific issue it might have other usage.

20.8 Passes

todo

```
\showmakeup[vpenalty,line]
\balancefinalpenalties 6 10000 9000 8000 7000 6000 5000\relax
\balancevsize 5\lineheight
```

```
\setbox\scratchbox\vbox{\dorecurse{1}{\samplefile{tufte}\footnote{!}\par}}
\vbalance\scratchbox
```

20.9 Passes

In LuaMetaT_EX the `_par_builder` has been extended with additional features (like `_orphan`, `_tod`).

```
% threshold
% tolerance
% looseness
% adjdemerits
% originalstretch
% emergencystretch
% emergencyfactor
% emergencypercentage
```

20.9 Colofon

Author	Hans Hagen
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com
	contextgarden.net
	ntg-context@ntg.nl

21 Lines

The background of the entire image is a solid brown color. Overlaid on this background is a repeating pattern of thin, dark gray diagonal lines. These lines are spaced evenly and run from the top-left towards the bottom-right at an angle of approximately 45 degrees.

low level

TEX

lines

Contents

21.1	Introduction	307
21.2	Warning	308
21.3	Constructing paragraphs	308
21.4	Subpasses	312
21.5	Definitions	316
21.6	Tracing	318
21.7	Criterion	321
21.8	Examples	321
21.9	Pages	322
21.10	Profiles	325

21.1 Introduction

There is no doubt that \TeX does an amazing job of “breaking paragraphs into lines” where a paragraph is a sequence of words in the input separated by spaces or its equivalents (single line endings turned space). The best descriptions of how that is done can be found in Don Knuths “The \TeX Book”, “ \TeX The Program” and “Digital Typography”. Reading and rereading the relevant portions of those texts is a good exercise in humility.

That said, whatever follows here builds upon what Knuth gave us and in no way we pretend to do better than that. It started out as a side track of improving rendering math in combination with more control over breaking inline math. It pretty much about having fun with the par builder but in the end can also help make your results look better. This is especially true for proze.

Trying to describe the inner working of the par builder makes no sense. Not only is it kind of complex, riddled with magic constants and heuristics, but there is a good chance for us to talk nonsense thanks to misunderstanding. However, some curious aspects will be brought up. Consider what follows a somewhat naive approach and whatever goes wrong, blame the authors, not \TeX .

If you're one of those reader who love to complain about the bad manuals, you can stop reading here. There is plenty said in the mentioned books but you can also consult Viktor Eijkhouts excellent “ \TeX by Topic” (just search the web for how to get these books). If you're curious and in for some adventure, keep reading.

21.2 Warning

This is a first version. What is described here will stay but is still experimental and how it evolves also depends on what demands we get from the users. We have defined some experimental setups in ConT_EXt. We will try to improve the explanations in ways

```
\setupalign[granular]
```

We will explain below what that means, but let us already now make clear that this will likely become the default! As far as we can see, due to the larger solution space, the inter-word spacing is more even but that also means that some paragraphs can become one line less or more.

21.3 Constructing paragraphs

There are several concepts at work when T_EX breaks a paragraph into lines. Here we assume

- The spaces between words can stretch or shrink. We don't want that to be too inconsistent (visible) between two lines. This is where the terms loose and tight come into play.
- Words can be hyphenated but we don't want that to happen too often. We also discourage neighboring lines to have hyphens. Hyphenating the (pre) final line is also sort of bad.
- We definitely don't want words to stick out in the margin. If we have to choose, stretching is preferred over shrinking. If spaces become too small words, start to blur.
- If needed glyphs can stretch or shrink a little in order to get rid of excessive spacing. But we really want to keep it minimal, and avoid it when possible. Usually we permit more stretch than shrink. Not all scripts (and fonts for that matter) might work well with this feature.
- As a last resort we can stretch spaces so that we get rid of any still sticking out word. When T_EX reports an overfull box (often a line) you have to pay attention!

When T_EX decides where to break and when to finish doing so it uses a system of penalties

Here we shortly summarize the parameters that play a role in calculating what T_EX calls the

```
\ruledhbox to 20 ts{left \hss right}
\ruledhbox to 40 ts{left \hss right}
\ruledhbox to 5 ts{left \hss right}
```

```

\ruledhbox to 5 ts{left      right}
\ruledhbox to 5 es{%
  left
  \hskip 1ts plus 0.5ts\relax
  middle
  \hskip 1ts plus 1.5ts\relax
  right%
}

```

These boxes show a bit what happens with spacing that can stretch or shrink. The first three cases are not bad because it's what we ask for with the wildcard `\hss.`²⁹

```

left.....right
left.....right
left
left_right
left.....middle.....right

```

\TeX will run over each paragraph at most three times. On each such run, it will choose different

The process is primarily controlled by these parameters:

- `\pretolerance`: This number determines the success of the first, not hyphenated pass. Often the value is set to the plain \TeX value of 100. If \TeX finds a possible division
- `\tolerance`: This number determines the success of the second, hyphenated pass. Often the value is set to the plain \TeX value of 200.
- `\emergencystretch`: This dimension kicks in when the second pass is not successful. In \ConTeXt we often set it to `2\bodyfontsize`.

When we are (in \ConTeXt speak) tolerant, we have a value of 3000, while `verytolerant` has

We now come to the other relevant parameters. You need to keep in mind that the demerits are made from penalty values that get squared which is why parameters with demerits in their name have high values: a penalty of 50 squared has to relate to a demerit of 5000, so we might have $2500 + 5000$ at some point.

The formula (most often) used to calculate the demerits d is

$$d = (l + b + p)^2 + e$$

²⁹ We use this opportunity to promote the new `ts` and `es` units.

Here l is the `\linepenalty`, set to 10 in plain, b is the badness of the line, and p is the penalty of the current break (for example, added by hyphenation, or by breaking an inline formula). The e stands for extra non-local demerits, that do not depend on only the current line, like the `\doublehyphendemerits` that is added if two lines in a row are hyphenated.

The badness reflects how the natural linewidth relates to the target width and uses a cubic function. A badness of zero is of course optimal, but a badness of 99 is pretty bad. A magic threshold is 12 (around that value a line is considered decent). If you look at the formula above you can now understand why the line penalty defaults to the low value of 10.

- `\hyphenpenalty`: When a breakpoint occurs at a discretionary this one gets added. In LuaMetaTeX we store penalties in the discretionary nodes but user defined `\discretionary` is not supported.
- `\linepenalty`: Normally this is set to 10 and it is the baseline for a breakpoint. This is again a small value compared to for instance the penalties that you find in inline math. There we need some breakpoints and after binary and relation symbols such an opportunity is created. The specific penalties are normally 500 and 700. One has to keep in mind, as shown in the formula above, that the penalties are not acting on a linear scale when the demerits are calculated. Math spacing and penalty control is discussed in the (upcoming) math manual.
- `\doublehyphendemerits`: Because it is considered bad to have two hyphens in a row this is often set pretty high, many thousands. These are treated as demerits (so outside of the squared part of the above formula).
- `\finalhyphendemerits`: The final (pre last) line having a hyphen is also considered bad. The last line is handled differently anyway, just because it gets normally flushed left.
- `\adjdemerits`: lines get rated in terms of being loose, decent, tight, etc. When two lines have a different rating we bump the total demerits.
- `\looseness`: it is possible to force less or more lines but to what extend this request is honored depends on for instance the possible (emergency) stretch in the spaces (or any glue for that matter).

It is worth noticing that you can set `\lastlinefit` such that the spaces in the last line will be comparable to those in the preceding line. This is a feature that ϵ -TeX brought.

The next one is a flag that triggers expansion (or compression) of glyphs to kick in. Those get added to the available stretch and/or shrink of a line:

Constructing paragraphs

- `\adjustspacing`: Its value determines if expansion kicks in: glyphs basically get a stretch and shrink value, something that helps filling our lines. We only have zero, two and three (and not the pdfTeX value of two): three means ‘only glyphs’ and two means ‘font kerns and glyphs’.

In LuaMetaTeX we also have:

- `\linebreakcriterion`: The normal distinction between loose, decent and tight in TeX uses 12 for 0.5 and 99 for about 1.0, but because we have more granularity (.25) we get it, hardly any user will change these values. One can think of the 100 squared becomes a 10000 (at least this helps relating these numbers) and 10000 is pretty bad in TeXs perception.
- `\adjustspacingstep`: When set this one is are used instead of the font bound value which permits local control without defining a new font instance.
- `\adjustspacingstretch`: idem.
- `\adjustspacingshrink`: idem.
- `\orphanpenalty`: This penalty will be injected before the last word of a paragraph.
- `\orphanpenalties`: Alternatively a series of penalties can be defined. This primitive expects a count followed by that number of penalties. These will be injected starting from the end.

The shape of a paragraph is determined by `\hangindent`, `\hangafter`, `\parshape` and `\parindent`. The width is controlled by `\hsize`, `\leftskip`, `\rightskip`. In addition there are `\parinitleftskip`, `\parinitrightskip`, `\parfillleftskip` and `\parfillrightskip` that control first and last lines.

We also have these:

- `\linebreakpasses`: When set to one, the currently set `\parpasses` will be applied.
- `\parpasses`: This primitive defined a set of sub passes that kick in when the second pass is finished. This basically opens up the par builder. It is still experimental and will be improved based upon user feedback. Although it is a side effect of improving the breaking of extensive mixes of math and text, it is also quite useful for text only (think novels).

In the next sections we will explain how these can improve the look and feel of what you typeset.

21.4 Subpasses

In `LuaTeX` and therefore also in `LuaMetaTeX` a paragraph is constructed in steps:

- The list of nodes that makes the paragraph is hyphenated: words become a mixture of glyphs and discretionaries.
- That list is processed by a font handler that can remove, add or change glyphs depending on how glyphs interact. This depends on the language and scripts used.
- The result is fed into the par builder that applies up to three passes as mentioned before.

In traditional `TeX` these three actions are combined into one and the overhead is shared. In

Here is an example of a setup. We set a low tolerance for the first pass and second pass. We can do that because we don't need to play safe nor need to compromise.

```
\pretolerance 75
\tolerance 150
\parpasses 3
  threshold 0.025pt
  classes \indecentparpassclasses
  tolerance 150
next
  threshold 0.025pt
  classes \indecentparpassclasses
  tolerance 200
  emergencystretch .25\bodyfontsize
next
  threshold 0.025pt
  classes \indecentparpassclasses
  tolerance 200
  optional 1
  emergencystretch .5\bodyfontsize
\relax
\linebreakpasses 1
```

Because we want to retain performance we need to test efficiently if we really need the (here upto three) additional passes, so let's see how it is done. When a pass list is defined, and line break passes are enabled, the engine will check *after* the second pass if

- `overflow` : the maximum value found, this is something really bad.
- `underflow` : the maximum value found, this is something we can live with.
- `verdict` : what is the worst badness of lines in this paragraph.
- `classified` : what classes are assigned to lines, think looseness, decent and tight.

There are two cases where the engine will continue with the applying passes: there is an overflow or there is a verdict (max badness) larger than zero. When we tested this on some large documents we noticed that this is nearly always true, but by checking we save a few unnecessary passes.

Next we test if a pass is really needed, and if not we check the next pass. When a pass is done, we pick up where we left, but we test for the overflow or badness every sub pass. The next checks make us run a pass:

- `overfull_exceeds_threshold`
- `verdict_exceeds_badness`
- `classified_overlaps_classes`

Here `threshold`, `badness` and `classes` are options in a pass section. Which test makes sense depends a bit on how `TeX` sees the result. Internally `TeX` uses numbers for its classification.

	indecent	almostdecent	loose	tight
1 veryloose	+	+	+	
2 loose	+	+	+	
4 semiloose	+		+	
8 decent				
16 semitight	+			+
32 tight	+	+		+

The `semiloose` and `semitight` values are something `LuaMetaTeX`. In `ConTeXt` we have these for

The sections in a par pass setup are separated by `next`. For testing purposes you can add `skip` and `quit`. The threshold tests against the `overfull` value, the `badness` against the `verdict` and `classes` checks for overlap with encountered classes, the classification.

You can specify an identifier in the first segment that then will be used in tracing but it is also passed to callbacks that relate to this feature. Discussing these callbacks is outside the scope of this wrapup.

You need to keep in mind that parameters are not reset to their original values between two subpasses of a paragraph. We have `tolerance` and `emergencystretch` which are handy for simple setups. When we start with a small tolerance we often need to bump that one. The stretch is likely a last resort. The usual demerits can be set too: `doublehyphendemerits`, `finalhyphendemerits` and `adjdemerits`. We have `extrahyphenpenalty` that gets added to the penalty in a discretionary. You can also set `linepenalty` to a different value than it normally gets.

The looseness can be set but keep in mind that this only makes sense in very special cases. It's hard to be loose when there is not much stretch or shrink available. The `linebreakcriterion` parameter can best be left untouched and is mostly there for testing purposes.

The LuaMetaTeX specific `orphanpenalty` gets injected before the last word in a paragraph.

The next four parameters are related to expansion: `adjustspacing`, `adjustspacing-step`, `adjustspacingshrink` and `adjustspacingstretch`. Here we have several scenarios.

- Fonts are set up for expansion (in ConTeXt for instance with the quality specifier). When
- When we don't enable it, the par pass can do it by setting `adjustspacing` (to 3).
- When the other parameters are set these will overload the ones in the font, but used with the factors in there, so different characters get scaled differently. You can set the step to one to get more granular results.
- When expansion is *not* set on the font, setting the options in a pass will activate expansion.

When a font is not set up to use expansion, you can do something like this:

```
\parpasses      6
  classes                \indecentparpassclasses
  threshold              0.025pt
  tolerance              250
  extrahyphenpenalty     50
  orphanpenalty          5000
% font driven
next ifadjustspacing
  threshold              0.025pt
  classes                \tightparpassclasses
  tolerance              300
  adjustspacing          3
```

Subpasses

```

    orphanpenalty      5000
next ifadjustspacing
  threshold            0.025pt
  tolerance            350
  adjustspacing        3
  adjustspacingstep    1
  adjustspacingshrink  20
  adjustspacingstretch 40
  orphanpenalty        5000
  emergencystretch     .25\bodyfontsize
% otherwise, factors 1000
next
  threshold            0.025pt
  classes              \tightparpassclasses
  tolerance            300
  adjustspacing        3
  adjustspacingstep    1
  adjustspacingshrink  10
  adjustspacingstretch 15
  orphanpenalty        5000
next
  threshold            0.025pt
  tolerance            350
  adjustspacing        3
  adjustspacingstep    1
  adjustspacingshrink  20
  adjustspacingstretch 40
  orphanpenalty        5000
  emergencystretch     .25\bodyfontsize
% whatever
next
  threshold            0.025pt
  tolerance            3000
  orphanpenalty        5000
  emergencystretch     .25\bodyfontsize
\relax

```

With `ifadjustspacing` you ignore steps that expect the font to be setup, so you don't waste time if that is not the case.

There is also a callback parameter but that one is experimental and used for special purposes and testing. We don't expect users to mess with that.

A really special feature is optional content. Here we use as example a quote from Digital Typography:

Many readers will skim over formulas on their first reading of your exposition. Therefore, your sentences should flow smoothly when all but the simplest formulas are replaced by \quotation {blah} or some other \optionalword {1} {grunting }noise.

Here the grunting (with embedded space) is considered optional. When you set \linebreakoptional to 1 this word will be typeset. However, when you set the pass parameter linebreakoptional to 0 it will be skipped. There can be multiple optional words with different numbers. The numbers are actually bits in a bit set so plenty is possible. However, normally these two values are enough, if used at all.

21.5 Definitions

The description above is rather low level and in practice users will use a bit higher level interface. Also, in practice only a subset of the parameters makes sense in general usage. It is not that easy to decide on what parameter subset will work out well but it can be fun to play with variants. After all, this is also what TeX is about: look, feel and

Some users praise the ability of recent TeX engines to provide expansion and protrusion. The MetaTeX that normally can be neglected), it also makes the output files larger. Some find it

The traditional (MkIV) way to set up expansion is to add this to the top of the document, or at least before fonts get loaded.

```
\scratchcounter 1
\bgroup
\advance\scratchcounter 1
\egroup
\the\scratchcounter
```

and later on to enable it with:

```
\setupalign[hz]
```

However, par passes make it possible to be more selective. Take the following two definitions:

```

\startsetups align:pass:quality:1
  \pretolerance 50
  \tolerance 150
  \parpasses 6
    identifier \parpassidentifier{quality:1}
    threshold 0.025pt
    tolerance 175
  next
    threshold 0.025pt
    tolerance 200
  next
    threshold 0.025pt
    tolerance 250
  next
    classes \almostdecentparpassclasses
    tolerance 300
    emergencystretch .25\bodyfontsize
  next ifadjustspacing
    classes \indecentparpassclasses
    tolerance 300
    adjustspacing 3
    emergencystretch .25\bodyfontsize
  next
    threshold 0.025pt
    tolerance 3000
    emergencystretch 2\bodyfontsize
  \relax
\stopsetups

```

```

\startsetups align:pass:quality:2
  \pretolerance 50
  \tolerance 150
  \parpasses 5
    identifier \parpassidentifier{quality:2}
    threshold 0.025pt
    tolerance 175
  next
    threshold 0.025pt
    tolerance 200
  next

```

```

        threshold      0.025pt
        tolerance      250
    next ifadjustspacing
        classes        \indecentparpassclasses
        tolerance      300
        adjustspacing   3
        emergencystretch .25\bodyfontsize
    next
        threshold      0.025pt
        tolerance      3000
        emergencystretch 2\bodyfontsize
\relax
\stopsetups

```

You can now enable one of these:

```
\setupalignpass[quality:1]
```

The result is shown in figure 21.1 where you can see that expansion is applied selectively;

21.6 Tracing

There are several ways to see what goes on. The engine has a tracing option that is set with `\tracingpasses`. Setting it to 1 reports the passes on the console, and a value of 2 also gives some details.

There is also a tracker, `paragraphs.passes` that can be enabled. This gives a bit more information:

```

\enabletrackers[paragraphs.passes]
\enabletrackers[paragraphs.passes=summary]
\enabletrackers[paragraphs.passes=details]

```

If you want to see where expansion kicks in, you can use:

```
\showmakeup[expansion]
```

This is just one of the options, spaces, penalties, glue are are useful when you play with passes, but if you are really into the low level details, this is what you want:

```

\startnarrower[5*right]
\startshowbreakpoints[option=margin,offset=\dimexpr{.5\emwidth-\rightskip}]

```



```
\samplefile{tufte}
\stopshowbreakpoints
\stopnarrower
```

```
We_thrive_in_information-thick_worlds_because_of_our_marvelous_and_
everyday_capacity_to_select,_edit,_single_out,_structure,_highlight,_group,_pair,_merge,_har-
monize,_synthesize,_focus,_organize,_condense,_reduce,_boil_down,_choose,_cat-
egorize,_catalog,_classify,_list,_abstract,_scan,_look_into,_idealize,_isolate,_dis-
criminate,_distinguish,_screen,_pigeonhole,_pick_over,_sort,_integrate,_blend,_in-
spect,_filter,_lump,_skip,_smooth,_chunk,_average,_approximate,_cluster,_ag-
gregate,_outline,_summarize,_itemize,_review,_dip_into,_flip_through,_browse,_glance_
into,_leaf_through,_skim,_refine,_enumerate,_glean,_synopsise,_winnow_
the_wheat_from_the_chaff_and_separate_the_sheep_from_the_goats._
```

You_can_see_the_chosen_solutions_with_

```
\showbreakpoints[n=1]
```

```
1  1  0  10001  0  verytight  disc
2  2  1  10001  0  verytight  disc
3  3  2  10001  0  verytight  disc
4  4  3  10001  0  verytight  disc
5  5  4  10001  0  verytight  disc
6  6  5  10001  0  verytight  disc
7  7  6  10001  0  verytight  disc
8  8  7  10001  0  verytight  disc
```

```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
6 5 4 3 2 1
7 6 5 4 3 2 1
8 7 6 5 4 3 2 1
```

```
pass      : 3  demerits : 0
subpass   : T  looseness : 0
subpasses : 0
```

When_we_started_playing_with_the_par_builder_in_the_perspective_of_math,_we_side_
tracked_and_ended_up_with_a_feature_that_can_ge_used_in_controlled_situations._Cur-
rently_we_only_have_a_low_level_ConT_EXt_interface_for_this_(see_figure_21.2)._

The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

```
\tracinglousiness 1 \lousiness 0
```

```
\lousiness 1 11 0
```

```
\silliness 11
```

Figure 21.2 Influencing the way T_EX breaks lines applied to ward.tex.

21.7 Criterion

The `granular_alignment_option` will configure the `linebreakcriterion` to work with 0.25 steps instead of 0.50 steps which means that successive lines can become a bit closer in spacing. There is no real impact on performance because testing happens anyway. In figure 21.3 you see some examples, where in some it indeed makes a difference.

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

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

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

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

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

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

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

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

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

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

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

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

Figure 21.3 More granular interline criteria.

21.8 Examples

The `ConTeXt` distribution comes with a few test setups: `spac-imp-tests.mkxl`. Once we have

Currently we provide the following predefined passes that you can enable with `\setupalignpass:decent,quality,test1,test2,test3,test4,test5`. We hope that users are willing to test these.

Criterion

21.9 Pages

While the par builder does multiple passes, the page builder is a single pass progressive routine. Every time something gets added to the (so called) main vertical list the page state gets updated and when the page overflows what has been collected gets passed to the output routine. It is to a large extend driven by glue (with stretch and shrink) and penalties and when content (boxes) is added the process is somewhat complicated by inserts as these needs to be taken into account too.

You can get pages that run from top to bottom by adding stretch between lines but by default in ConT_EXt we prefer to fill up the bottom with white space.

It can be hard to make decisions at the T_EX end around a potential page break because in

Penalties play an important role and because these are used to control for instance widows and clubs high values can lead to underfull pages so if we want to influence that we need to cheat. For this we have three experimental mechanisms:

- tweaking the page goal: `\pageextragoal`
- initializing the state quantities: `\initialpageskip`
- adapting the state quantities as we go: `\additionalpageskip`

The first tweak is for me to play with, and when a widow or club is seen the extra amount can kick in. This feature is likely to be replaced by a more configurable one.

The second tweak lets the empty page start out with some given height, stretch and shrink. This variable is persistent over pages. This is not true for the third tweak: it kicks in when the page gets initialized or as we go, but after it has been applied the value

Adapting the layout (within the regular text area) is done with `\setpagelooseness` as demonstrated in figure 21.4 and figure 21.5. Possible parameters are `lines`, `height`, `str`



Figure 21.4 Cheating with page dimensions: `[lines=2]`.

It is not that trivial to fulfill the wide range of user demands but over time the `\setupalign` commands has gotten plenty of features. Getting for instance windows and clubs right in the kind of mixed usage that is common in ConT_EXt is not always easy. One o

The examples in figure 21.6, 21.7 and 21.8 scale vertically in order to fill up the text area;

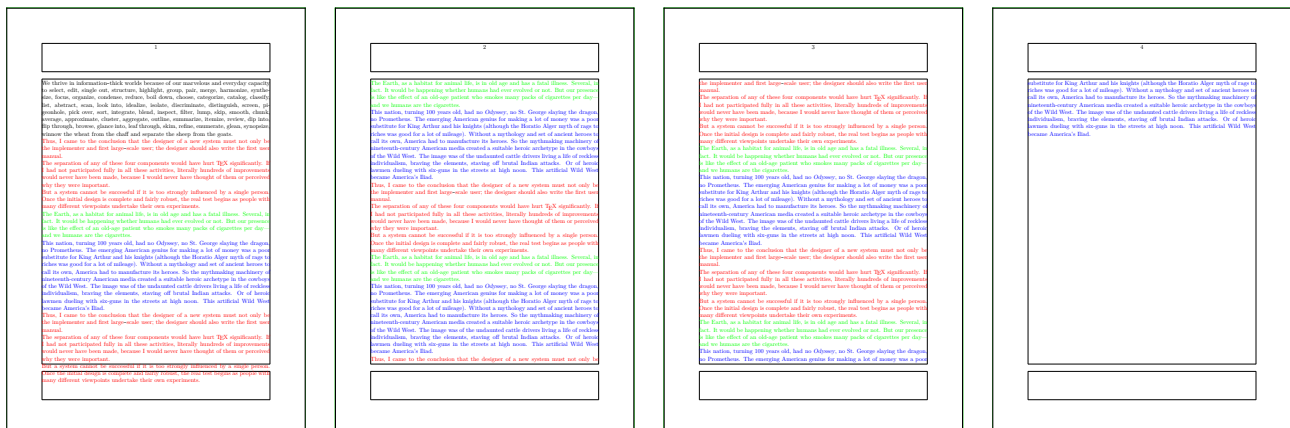


Figure 21.5 Cheating with page dimensions: [-3].

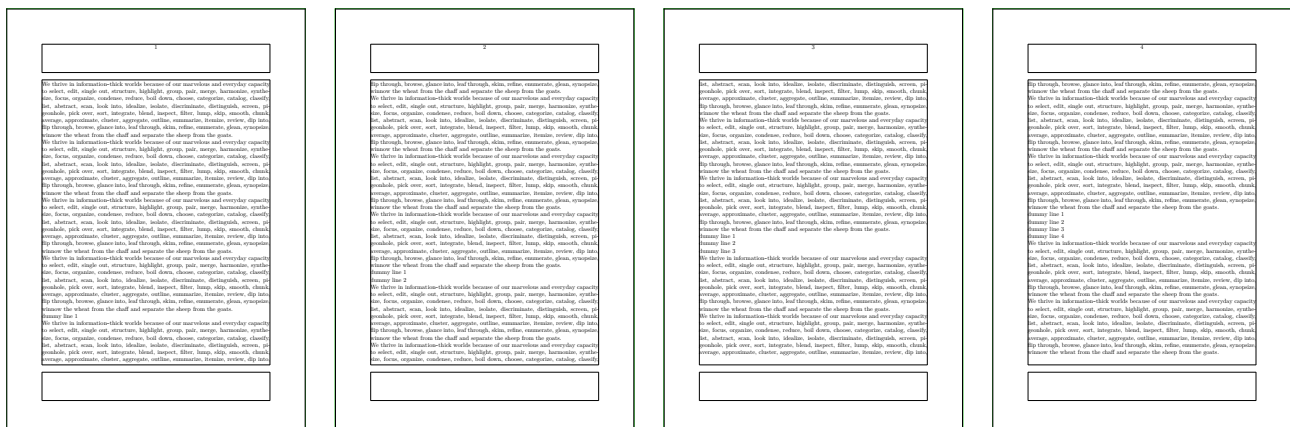


Figure 21.6 Cheating with vertical expansion: [vz=no].

There are a few other tweaks but these one can wonder about these. We can add stretch and shrink to the baseline skip, something that can also be triggered with the 'spread' option to \setupalign, assuming that also height is given). An alternative is to permit an extra line and accept a visual overflow, assuming that the layout is set up to make sure that the footer line doesn't overlap. None of this guarantees that a whole document with plenty of graphics and special constructs will come out well, but for text only it might work okay. Figures 21.9, 21.10 and 21.11 show some of

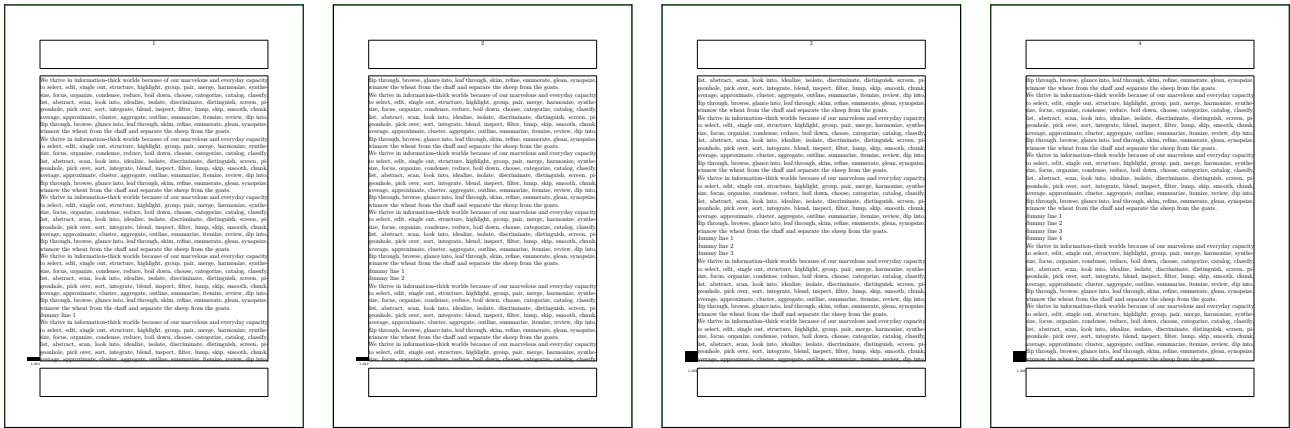


Figure 21.8 Cheating with vertical expansion: [vz=2].

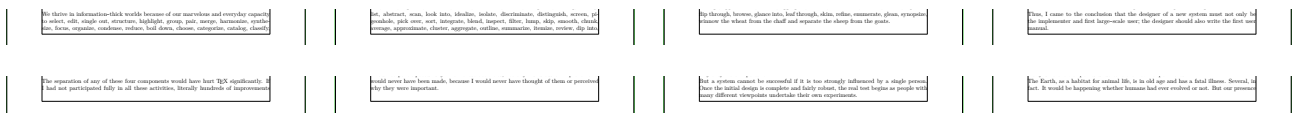


Figure 21.9 Cheating: just high penalties.

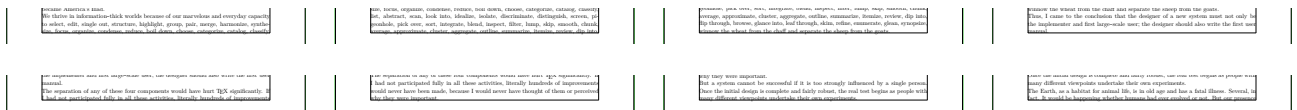


Figure 21.10 Cheating: `\baselineskip 1\baselineskip plus 1pt minus .1pt`.

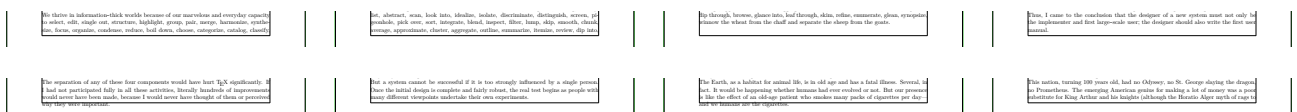


Figure 21.11 Cheating: `\pageextragoal\lineheight`.

21.10 Profiles

You can have a paragraph with lines that exceed the maximum height and/or depth or where spaces end up in a way that create so called rivers. Rivers are more a curiosity than an annoyance because any attempt to avoid them is likely to result in a worse looking result. The unequal line distances can be annoying too but these can be avoided when bringing lines closer together doesn't lead to clashes. This can be done without reformatting the paragraph by passing the profile option to `\setupalign`. It comes at the cost of a little more runtime and (as far as we observed) it kicks in seldom, for instance when inline math is used that has super- or subscripts, radicals, fractions or other slightly higher constructs.

21.10 Colofon

Author	Hans Hagen & Mikael Sundqvist
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com contextgarden.net ntg-context@ntg.nl

22 Debugging

low level

TEX

debugging

Contents

22.1	Introduction	328
22.2	Token lists	328
22.3	Node lists	331
22.4	Visual debugging	332
22.5	Math	333
22.6	Fonts	335
22.7	Overflow	336
22.8	Side floats	337
22.9	Struts	338
22.10	Features	338
22.11	Profiling	339
22.12	Par builder	340
22.13	More	341

22.1 Introduction

Below there will be some examples of how you can see what \TeX is doing. We start with some \MetaTeX engine. More details about what is possible can be found in documents in the \ConTeXt distribution, for instance the ‘lowlevel’ manuals.

Typesetting involves par building, page building, inserts (footnotes, floats), vertical adjusters (stuff before and after the current line), marks (used for running headers and footers), alignments (to build tables), math, local boxes (left and right of lines), hyphenation, font handling, and more and each has its own specific ways of tracing, either provided by the engine, or by \ConTeXt itself. You can run `context --trackers` to get a list of what \ConTeXt can do, as it lists most of them. But we start with the language

22.2 Token lists

There are two main types of linked lists in \TeX : token lists and node lists. Token lists relate \TeX and \LuaMetaTeX .

When you define a macro, like the following, you get a token list:

```
\def\test#1{\bgroup\bf#1\egroup}
```

Internally the `\test` macro has carry the argument part and the body, and each is encoded as a number plus a pointer to the next token.

control sequence: test

535205	19	49	match	argument 1
595462	20	0	end match	

598470	1	123	left brace	bgroup
597545	143	0	protected call	bf
600623	21	1	parameter reference	
596414	2	125	right brace	egroup

Here the first (large) number is a memory location that holds two 4 byte integers per token: the so called info part codes the command and sub command, the two smaller numbers in the table, and a link part that points to the next memory location, here the next row. The last columns provide details. A character like 'a' is one token, but a control sequence like \foo is also one token because every control sequence gets a number. So, both take eight bytes of memory which is why a format file can become large and memory consumption grows the more macros you use.

In the body of the above \test macro we used \bf so let's see how that looks:

permanent protected control sequence: bf

628	137	24	if test	ifmmode
629	131	0	expand after	expandafter
630	143	0	protected call	mathbf
631	137	3	if test	else
632	131	0	expand after	expandafter
633	143	0	protected call	normalbf
634	137	2	if test	fi

Here the numbers are much lower which is an indication that they are likely in the format. They are also ordered, which is a side effect of LuaMetaTeX making sure that the t

```
\tolerant\permanent\protected\def\test[#1]#:#2%
  {\iftok{#1}{sl}\bs\else\bf\fi#2}}
```

Gives us:

permanent tolerant protected control sequence: test

23056	12	91	other char	[U+0005B	
619411	19	49	match		argument 1
619343	12	93	other char] U+0005D	
597153	19	58	match		argument :

479011	19	50	match	argument 2
600584	20	0	end match	
618617	1	123	left brace	
610075	137	29	if test	iftok
597385	1	123	left brace	
250404	21	1	parameter reference	
598692	2	125	right brace	
599371	1	123	left brace	
593929	11	115	letter	s U+00073
618603	11	108	letter	l U+0006C
601395	2	125	right brace	
600505	143	0	protected call	bs
50580	137	3	if test	else
596159	143	0	protected call	bf
613899	137	2	if test	fi
618616	21	2	parameter reference	
600095	2	125	right brace	

If you are familiar with \TeX and spend some time looking at this you will start recognizing 115 translates to letter s because 11 is the so called command code of letters (also its `\catcode`) and the s has utf8 value 115. The LuaMeta \TeX specific `\iftok` conditional has

There is more to tell about these commands and the way macros are defined, for instance `tolerant` here means that we can omit the first argument (between brackets) in which case we pick up after the `#:`. With `protected` we indicate that the macro will not expand in for instance an `\edef` and `permanent` marks the macro as one that a user cannot redefine (assuming that overload protection is enabled). The extended macro argument parsing features and macro overload protection are something specific to LuaMeta \TeX .

These introspective tables can be generated with:

```
\luatokenable\test
```

after loading the module `system-tokens`. The reason for having a module and not a built-in tracer is that users seldom want to do this. Instead they might use `\showluatokens\test` that just reports something similar to the console and/or log file.

There is much more to tell but most users have no need to look into these details unless they are curious about what \TeX does. In that case using `tracingall` and `inspecting` Meta \TeX we have tried to improve these traces a bit but that's of course subjective and even

22.3 Node lists

A node list is what you get from input that is (to be) typeset. There are several ways to see what node lists are produced but these are all very verbose. Take for instance:

```
\setbox\scratchbox\hbox{test \bf test}
```

```
\showboxhere\scratchbox
```

This gives us:

```
\hlist[box][color=1,colormodel=1,mathintervals=1], width 47.8457pt, height 7.48193pt, depth
0.15576pt, direction l2r, state 1
.\list
..\glyph[unset][color=1,colormodel=1], protected, wd 4.42041pt, ht 7.48193pt, dp 0.15576pt, language
(n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <1: DejaVuSerif @ 11.0pt>, glyph U+0074
..\glyph[unset][color=1,colormodel=1], protected, wd 6.50977pt, ht 5.86523pt, dp 0.15576pt, language
(n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <1: DejaVuSerif @ 11.0pt>, glyph U+0065
..\glyph[unset][color=1,colormodel=1], protected, wd 5.64502pt, ht 5.86523pt, dp 0.15576pt, language
(n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <1: DejaVuSerif @ 11.0pt>, glyph U+0073
..\glyph[unset][color=1,colormodel=1], protected, wd 4.42041pt, ht 7.48193pt, dp 0.15576pt, language
(n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <1: DejaVuSerif @ 11.0pt>, glyph U+0074
..\glue[spaceskip][color=1,colormodel=1] 3.49658pt plus 1.74829pt minus 1.16553pt, font 1
..\glyph[unset][color=1,colormodel=1], protected, wd 5.08105pt, ht 7.48193pt, dp 0.15576pt, language
(n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <10: DejaVuSerif-Bold @ 11.0pt>, glyph
U+0074
..\glyph[unset][color=1,colormodel=1], protected, wd 6.99854pt, ht 5.86523pt, dp 0.15576pt, language
(n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <10: DejaVuSerif-Bold @ 11.0pt>, glyph
U+0065
..\glyph[unset][color=1,colormodel=1], protected, wd 6.19287pt, ht 5.86523pt, dp 0.15576pt, language
(n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <10: DejaVuSerif-Bold @ 11.0pt>, glyph
U+0073
..\glyph[unset][color=1,colormodel=1], protected, wd 5.08105pt, ht 7.48193pt, dp 0.15576pt, language
(n=1,l=2,r=3), hyphenationmode "79F3F, options "80, font <10: DejaVuSerif-Bold @ 11.0pt>, glyph
U+0074
```

The periods indicate the nesting level and the slash in front of the initial field is mostly a historic curiosity because there are no `\hlist` and `\glue` primitives, but actually there is in LuaMetaTeX a `\glyph` primitive but that one definitely doesn't want the shown a

That said, here we have a horizontal list where the list field points to a glyph that itself points to a next one. The space became a glue node. In LuaTeX and even more in LuaMetaTeX all nodes have or get a subtype assigned that indicates what we're dealing with. T

It will be clear that watching a complete page, with many nested boxes, rules, glyphs, discretionaries, glues, kerns, penalties, boundaries etc quickly becomes a challenge which is why we have other means to see what we get so let's move on to that now.

22.4 Visual debugging

In the early days of ConT_EXt, in the mid 90's of the previous century, one of the first present T_EXt MkIV it was implemented in a different (less intrusive) way and it got gradually extended

```
\showmakeup \hbox{test \bf test}
```

This gives us a framed horizontal box, with some text and a space glue:

H **test test**

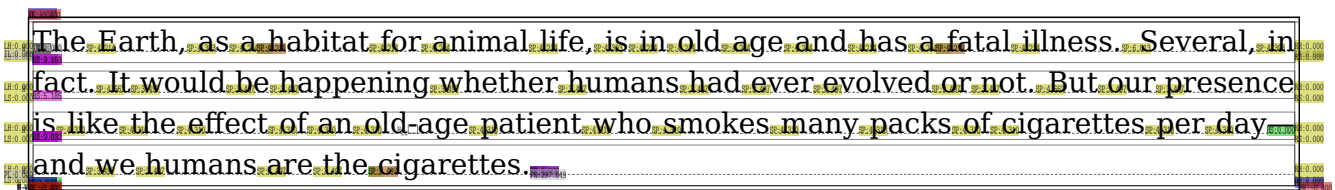
Of course not all information is well visible simply because it can be overlayed by what follows, but one gets the idea. Also, when you have a layer capable pdf viewer you can turn on and off categories, so you can decide to only show glue. You can also do that immediately, with `\showmakeup[glue]`.

There is a lot of granularity: `hbox`, `vbox`, `vtop`, `kern`, `glue`, `penalty`, `fontkern`, `strut`, `what-sit`, `glyph`, `simple`, `simplehbox`, `simplevbox`, `simplevtop`, `user`, `math`, `italic`, `origin`, `discretionary`, `expansion`, `line`, `space`, `depth`, `marginkern`, `mathkern`, `dir`, `par`, `mathglue`, `mark`, `insert`, `boundary`, the more selective `vkern`, `hkern`, `vglue`, `hglue`, `vpenalty` and `hpenalty`, as well as some presets like `boxes`, `makeup` and `all`.

When we have:

```
\showmakeup \framed[align=normal]{\samplefile{ward}}
```

we get:



And that is why exploring this with a layers enabled pdf viewer can be of help. Alternatively a more selective use of `\showmakeup` makes sense, like

```
\showmakeup[line,space] \framed[align=normal]{\samplefile{ward}}
```

Here we only see lines, regular spaces and spaces that are determined by the space factor that is driven by punctuation.

The Earth, as a habitat for animal life, is in old age and has a fatal illness. Several, in fact. It would be happening whether humans had ever evolved or not. But our presence is like the effect of an old-age patient who smokes many packs of cigarettes per day—and we humans are the cigarettes.

We can typeset the previous example with these settings:

```
\leftskip      2cm
\rightskip     3cm
\hangindent    1cm
\hangafter     2
\parfillrightskip 1cm
\parfillleftskip 1cm % new
\parinitrightskip 1cm % new
\parinitleftskip 1cm % new
\parindent     2cm % different
```

This time we get:

The Earth, as a habitat for animal life, is in old age
and has a fatal illness. Several, in fact. It would be happening
whether humans had ever evolved or not. But our pres-
ence is like the effect of an old-age patient who smokes
many packs of cigarettes per day—and we humans are
the cigarettes.

Looking at this kind of output only makes sense on screen where you can zoom in but what we want to demonstrate here is that in LuaMetaTeX we have not only a bit more TeX and LuaMetaTeX we have to take care of that.

Another characteristic is that the paragraph stores these (and many more) properties in the so called initial par node so that they work well in situations where grouping would interfere with our objectives. As with all extensions, these are things that can be configured in detail but they are enabled in ConTeXt by default.

22.5 Math

Math is a good example where this kind of tracing helps development. Here is an example:

```
\im { \showmakeup y = \sqrt {2x + 4} }
```

Scaled_up_we_get:_

$$y = \sqrt{2x + 4}$$

Instead_of_showing_everything_we_can_again_be_more_selective:_

```
\im {
  \showmakeup[mathglue,glyph]
  y = \sqrt {2x + 4}
}
```

Here_we_not_only_limit_ourselves_to_math_glue,_but_also_enable_showing_the_bound-
ing_boxes_of_glyphs._

$$y = \sqrt{2x + 4}$$

This_example_also_shows_that_in_LuaMetaTeX_we_have_more_classes_than_in_a_traditional_TeX

```
\im {
  \showmakeup[mathglue,glyph]
  y = \sqrt {x__1__a {\darkred +} x__1__b}
}
```

$$y = \sqrt{x_1 a + x_1 b}$$

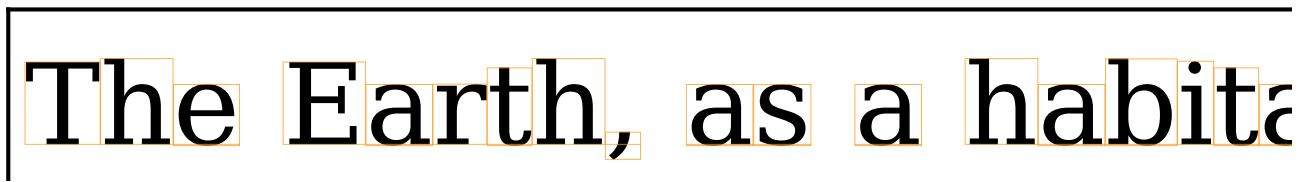
Here_the_variable_class_is_used_for_alphabetic_characters_and_some_more,_contrary_
to_the_more_traditional_(often_engine_assigned)_ordinary_class_that_is_now_used_for_
the_left-overs._

22.6 Fonts

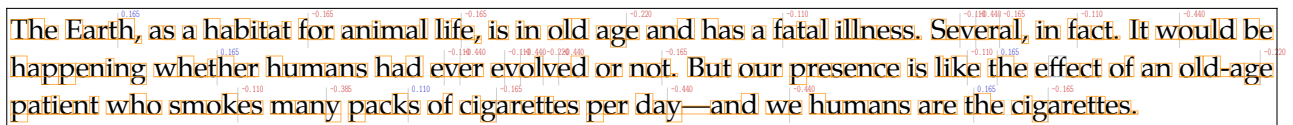
Some_of_the_mentioned_tracing_has_shortcuts,_for_instance_\showglyphs._Here_we_show_the_same_sample_paragraph_as_before:_

```
\showglyphs
\showfontkerns
\framed[align=normal]{\samplefile{ward}}
```

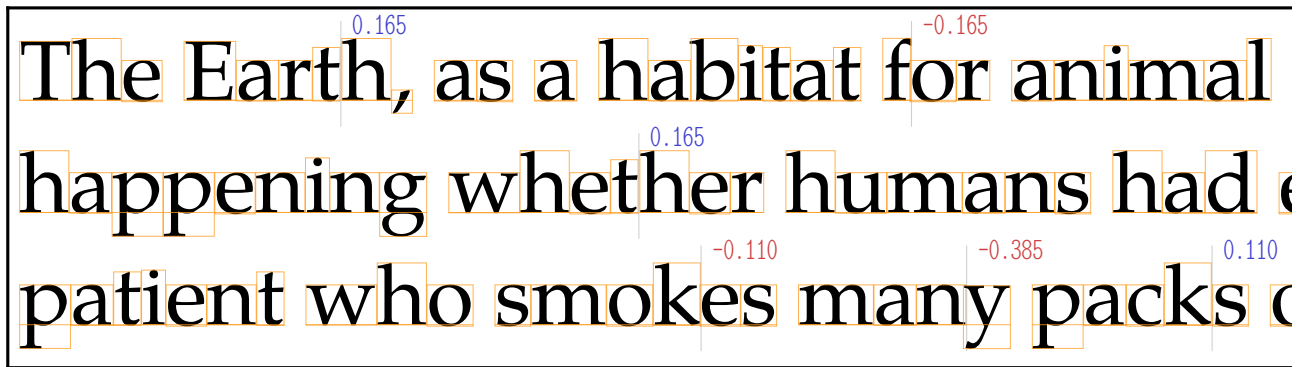
Here_is_the_upper_left_corner_of_the_result:_



What_font_kerns_we_get_depends_on_the_font,_here_we_use_pagella:_

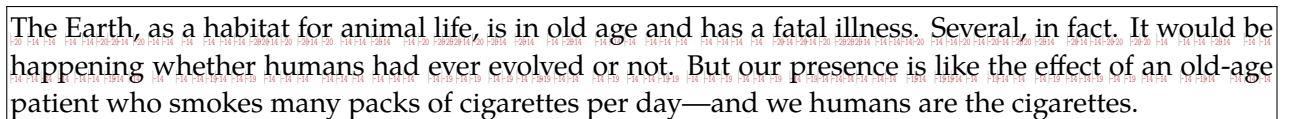


If_we_zoom_in_the_kerns_are_more_visible:_



And_here_is_another_one:_

```
\showfontexpansion
\framed[align={normal,hz}]{\samplefile{ward}}
```



or_blown_up:_

The Earth, as a habitat for animal
happening whether humans had €
patient who smokes many packs c

The_last_line_(normally)_doesn't_need_expansion,_unless_we_want_it_to_compatible_with_preceding_lines,_space-wise._So_when_we_do_this:_

```
\showfontexpansion
\framed[align={normal,hz,fit}]{\samplefile{ward}}
```

the_fit_directives_results_in_somewhat_different_results:_

The Earth, as a habitat for animal
happening whether humans had €
patient who smokes many packs

As_with_other_visual_tracers_you_can_get_some_insight_in_how_T_EX_turns_your_input_into_a_ty

22.7 Overflow

By_default_the_engine_is_a_bit_stressed_to_make_paragraphs_fit_well._This_means_that_we_can_get_overflowing_lines._Because_there_is_a_threshold_only_visible_overflow_is_reported._If_you_want_a_visual_clue,_you_can_do_this:_

```
\enabletrackers[builders.hpack.overflow]
```

With:_

```
\ruledvbox{\hsize 3cm test test test test test test test test}
```

We get:

```
test_test_test
test_test_test
test_test
```

The red bar indicates a potential problem. We can also get an underflow, as demonstrated here:

```
\ruledvbox {
  \setupalign[verytolerant,stretch]
  \hsize 3cm test test test test test test test test
}
```

Now we get a blue bar that indicates that we have a bit more stretch than is considered optimal:

```
test_test
test_test
test_test
test_test
```

Especially in automated flows it makes sense to increase the tolerance and permit stretch. Only when the strict attempt fails that will kick in.

22.8 Side floats

Some mechanisms are way more complex than a user might expect from the result. An example is the placement of float and especially side floats.

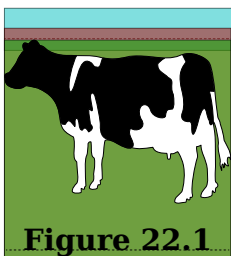


Figure 22.1

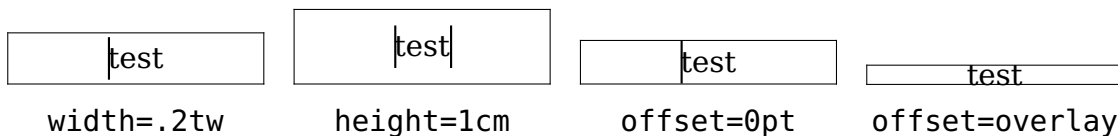
Not only do we have to make sure that the spacing before such a float is as good and consistent as possible, we also need the progression to work out well, that is: the number of lines that we need to indent.

For that we need to estimate the space needed, look at the amount of space before and after the float, check if it will fit and move to the next page if needed. That all involves dealing with interline spacing, interparagraph spacing, spacing at the top of a page, permitted slack at the bottom of page, the depth of the preceding lines, and so on. The tracer shows some of the corrections involved but leave it to the user to imagine what it relates to; the previous sentence gives some clues. This tracker is enabled with:

```
\enabletrackers[floats.anchoring]
```

22.9 Struts

We now come to one of the most important trackers, `\showstruts`, and a few examples shows why:



Here in all cases we've set the width to 20 percent of the text width (tw is an example of a plugged in dimension). In many places ConTeXt adds struts in order to enforce p

22.10 Features

Compared to the time when T_EX showed up the current fonts are more complicated, especially

```
font      22: _texgyrepagella-regular.otf @ 10.0pt
```

```
features  [basic: _kern=yes, _liga=yes, _mark=yes, _mkmk=yes, _script=dflt] [extra: _ana
```

```
step_1    effe_fietsen U+65:e U+66:f [ pre: U+2D:␣ ] U+66:f U+65:e [ pre: U+5F:_
           replace: U+5F:_ ] U+66:f U+69:i U+65:e U+74:t [ pre: U+2D:␣ ] U+73:s
           U+65:e U+6E:n
           feature 'liga', _type 'gsub_ligature', _lookup 's_s_9', _replacing U+66_(f
```

```
step_2    effe_fietsen U+65:e [ pre: U+66:f U+2D:␣ post: U+66:f replace:
           U+FB00:ff ] U+65:e [ pre: U+5F:_ replace: U+5F:_ ] U+66:f U+69:i
           U+65:e U+74:t [ pre: U+2D:␣ ] U+73:s U+65:e U+6E:n
           feature 'liga', _type 'gsub_ligature', _lookup 's_s_10', _replacing U+66_(f
```

```
step_3    effe_fietsen U+65:e [ pre: U+66:f U+2D:␣ post: U+66:f replace:
           U+FB00:ff ] U+65:e [ pre: U+5F:_ replace: U+5F:_ ] U+FB01:ff U+65:e
           U+74:t [ pre: U+2D:␣ ] U+73:s U+65:e U+6E:n
           feature 'kern', _type 'gpos_pair', _lookup 'p_s_0', _inserting_move_-0.1499
```

```
result    effe_fietsen U+65:e [ pre: U+66:f U+2D:␣ post: U+66:f [kern] replace:
           U+FB00:ff ] U+65:e [ pre: U+5F:_ replace: U+5F:_ ] U+FB01:ff U+65:e
           U+74:t [ pre: U+2D:␣ ] U+73:s U+65:e U+6E:n
```

Not all features listed here are provided by the font (only the four character ones) because we're using T_EX which, it being T_EX, means that we have plenty more ways to mess around.

With Cambria we get a single replacement combined with kerning:

```
font      23: _cambria.ttc @ 10.0pt

features  [basic: _kern=yes, _liga=yes, _mark=yes, _mkmk=yes, _script=latn] [extra: _ana

step_1    effe fietsen U+65:e U+66:f [ pre: U+2D:□ ] U+66:f U+65:e [ pre: U+5F:_
replace: U+5F:_ ] U+66:f U+69:i U+65:e U+74:t [ pre: U+2D:□ ] U+73:s
U+65:e U+6E:n
feature '_liga', _type 'gsub_contextchain', _chain_lookup 's_s_38', _replaci

step_2    effe fietsen U+65:e U+66:f [ pre: U+2D:□ ] U+66:f U+65:e [ pre: U+5F:_
replace: U+5F:_ ] U+F016C:f U+69:i U+65:e U+74:t [ pre: U+2D:□ ]
U+73:s U+65:e U+6E:n
feature '_kern', _type 'gpos_pair', _merged_lookup 'p_s_0', _inserting_move

result    effe fietsen U+65:e U+66:f [ pre: U+2D:□ ] U+66:f [kern] U+65:e [ pre:
U+5F:_ replace: U+5F:_ ] U+F016C:f U+69:i U+65:e U+74:t [ pre:
U+2D:□ ] U+73:s U+65:e U+6E:n
```

One complication is that hyphenation kicks in which means that whatever we do has to take the pre, post and replacement bits into account combined which what comes before and after. Especially for complex scripts this tracker can be illustrative but even then only for those who like to see what fonts do and/or when they add additional features runtime.

22.11 Profiling

There are some features in ConT_EXt that are nice but only useful in some situations. An example

The command `\binom` is the standard notation for binomial coefficients and is preferred over `\choose`, which is an older macro that has limited compatibility with newer packages and font encodings: $|A| = \binom{N}{k}^2$. Additionally, `\binom` uses proper spacing and size for the binomial symbol. In conclusion, it is recommended to use `\binom` instead of `\choose` in T_EX for typesetting binomial coefficients.

The previous paragraph is what comes out by default, while the next one used these settings plus an additional `\enabletrackers[profiling.lines.show]`.

The command `\binom` is the standard notation for binomial coefficients and is preferred over `\choose`, which is an older macro that has limited compatibility with newer packages and font encodings: $|A| = \binom{N}{k}^2$. Additionally, `\binom` uses proper spacing and size for the binomial symbol. In conclusion, it is recommended to use `\binom` instead of `\choose` in TeX for typesetting binomial coefficients.

This feature will bring lines together when there is no clash and is mostly of use when a lot of inline math is used. However, when this variant of profiling (we have an older one too) is enabled on a 300 page math book with thousands of formulas, only in a few places it demonstrated effect; it was hardly needed anyway. So, sometimes tracing shows what makes sense or not.

22.12 Par builder

Here is a sample paragraph from Knuths "Digital Typography":

15. (This procedure maintains four integers (A, B, C, D) with the invariant meaning that "our remaining job is to output the continued fraction for $(Ay + B)/(Cy + D)$, where y is the input yet to come.") Initially set $j \leftarrow k \leftarrow 0$, $(A, B, C, D) \leftarrow (a, b, c, d)$; then input x_j and set $(A, B, C, D) \leftarrow (Ax_j + B, Ax_j + D, Cx_j + D, C)$, $j \leftarrow j + 1$, one or more times until $C + D$ has the same sign as C . (When $j > 1$ and the input has not terminated, we know that $1 < y < \infty$; and when $C + D$ has the same sign as C we know therefore that $(Ay + B)/(Cy + D)$ lies between $(A + B)/(C + D)$ and A/C .) Now comes the general step: If no integer lies strictly between $(A + B)/(C + D)$ and A/C , output $X_k \leftarrow \lfloor A/C \rfloor$, and set $(A, B, C, D) \leftarrow (C, D, A - X_k C, B - X_k D)$, $k \leftarrow k + 1$; otherwise input x_j and set $(A, B, C, D) \leftarrow (Ax_j + B, Ax_j + D, Cx_j + D, C)$, $j \leftarrow j + 1$. The general step is repeated ad infinitum. However, if at any time the final x_j is input, the algorithm imitates $B)/(Cx_j + D)$, using Euclid's algorithm, and terminates.

There are indicators with tiny numbers that indicate the possible breakpoints and we can see what the verdict is:

1	1	0	10001	0	verytight	disc	7	1	10	9	10001	80346	verytight	disc	14	12	11	10	9	8	6	3	2	1		
2	2	2	1	10001	0	verytight	disc	8	2	11	10	0	102946	decent	disc	15	13	11	10	9	8	6	3	2	1	
3	1	3	2	165	53125	veryloose	disc	9	2	12	11	16	116122	loose	disc	16	12	11	10	9	8	6	3	2	1	
	2	4	2	6	22756	decent	disc		2	13	11	0	115546	decent	disc	17	13	11	10	9	8	6	3	2	1	
4		5	4	166	76232	veryloose	disc	10		14	12	131	148503	veryloose	disc	18	17	13	11	10	9	8	6	3	2	1
	1	6	3	166	67650	loose	disc			15	13	14	128622	loose	disc											
		7	4	16	35932	tight	disc			16	12	14	128722	decent	disc	pass				3	demerits					
5		8	6	4	80346	decent	disc			17	13	28	129490	tight	disc	subpass				T	looseness					
6		9	8	10001	80346	verytight	disc	11		18	17	10001	129490	verytight	penalty	subpasses										

The last lines in the last column show the route that the result takes. Without going into details, here is what we did:

`\startshowbreakpoints`

```

\samplefile{math-knuth-dt}
\stopshowbreakpoints

\showbreakpoints

```

This kind of tracing is part of a mechanism that makes it possible to influence the choice by choosing a specific preferred breakpoint but that is something the average user is unlikely to do. The main reason why we have this kind of trackers is that when developing the new multi-step par builder feature we wanted to see what exactly it did influence. That mechanism uses an LuaMetaTeX feature where we can plug in a or badness are met. Each step can set the relevant parameters differently, including expansion, which actually makes for more efficient output and better runtime when that features is not needed to get better results.

22.13 More

There are many more visual trackers, for instance `layout.vz` for when you enabled vertical expansion, `typesetters.suspects` for identifying possible issues in the input like invisible spaces. Trackers like `nodes.destinations` and `nodes.references` will show the areas used by these mechanisms. There are also trackers for positions, (cjk and other), script handling, rubies, tagging, italic correction, breakpoints and so on. The examples in the previous sections illustrate what to expect and when to use a specific mechanism knowing this might trigger you to check if a tracker exists. Often the test suite has examples of usage.

22.13 Colofon

Author	Hans Hagen & Mikael Sundqvist
ConTeXt	2025.02.12 08:44
LuaMetaTeX	2.11.06 20250218
Support	www.pragma-ade.com contextgarden.net ntg-context@ntg.nl

23 Pages



low level

TEX

pages

Contents

23.1 Introduction	343
23.2 Rows becoming columns	343

23.1 Introduction

There are several builder in the engine: paragraphs, math, alignments, boxes and if course pages. But where a paragraph is kind of complete and can be injected on a line by line basis, a page is less finished. When enough content is collected the result so far is handled over to the output routine. Calling it a routine is somewhat confusing because it is not really a routine, it's the token list `\output` that gets expanded and what in there is supposed to something with the result, like adding inserts (footnotes, moved around graphics aka floats, etc.), adding headers and footers, possibly using marks, and finally wrapping up and shipping out.

The engine primarily offers a single column page so two or more columns are done by using tricks, like typesetting on a double height and splitting the result. If columns need to be balanced some extra work has to be done, and it's definitely non trivial when we have more than just text.

In this chapter we will discuss and collect some mechanisms that deal with pages or operate at the outer vertical level. We might discuss some primitive but more likely you will see various solutions based on `TeX` macros and Lua magic.

This is work in progress.

23.2 Rows becoming columns

This is an experimental mechanism. We need to check/decide how to deal with penalties. We also need to do more checking.

Conceptually this is a bit strange feature but useful nevertheless. There are several multi-column mechanisms in `ConTeXt` and each is made for a specific kind of usage. You can

```
\starttabulate[|l|l|]
\NC 1 \NC one    \NC \NR
\NC 2 \NC two    \NC \NR
\NC 3 \NC three  \NC \NR
\NC 4 \NC four   \NC \NR
\NC 5 \NC five   \NC \NR
```

`\stoptabulate`

but_you_don't_want_to_waste_space._So_you_might_want:_

1	one	4	four
2	two	5	five
3	three		

or_maybe_even_this:_

1	one	3	three	5	five
2	two	4	four		

but_still_wants_to_code_like_this:_

```
\starttabulate[|l|l|]
\NC 1 \NC one \NC \NR
\NC 2 \NC two \NC \NR
\NC 3 \NC three \NC \NR
\NC 4 \NC four \NC \NR
\NC 5 \NC five \NC \NR
\stoptabulate
```

You_can_do_this:_

```
\startcolumns[n=3]
\getbuffer
\stopcolumns
```

The_(mixed)_columns_mechanism_used_here_normally_works_ok_but_because_of_the_way_columns_are_packaged_they_don't_work_well_with_for_instance_'vz'. Page_columns_do_a_better_job_but_don't_mix_with_single_columns_that_well. Another_solution_is_this:_

```
\startrows[n=3,before=\blank,after=\blank]
\getbuffer
\stoprows
```

Here_the_result_is_collected_in_a_vertical_box,_post_processed_and_flushed_line_by_line. We_need_to_explicitly_handle_the_before_and_after_spacing_here_because_it_gets_discarded_(if_added_at_all). When_a_slice_of_the_box_is_part_of_the_shipped_out_page_the_cells_are_swapped_so_that_instead_of_going_horizontal_we_go_vertical. Compare_the_original_

1	one	2	two	3	three
---	-----	---	-----	---	-------

Rows becoming columns

4	four	5	five
---	------	---	------

with the swapped one:

1	one	3	three	5	five
2	two	4	four		

This is not really a manual but let's mention a few configuration options. The `n` parameter controls the number of columns. In order to support swapping this mechanism adds empty pseudo cells for as far as needed. By default the order is vertical but one can set it to horizontal instead. In the next example we have set `height` to 2\strutht

1	one	3	three	5	five
2	two	4	four		

When you set `height` and `depth` to `max` all cells will get these dimensions from the tallest cell. Compare:

1	$y = x + 1$	3	$y = \sqrt{x^2} + 1$
2	$y = x^2 + 1$	4	$y = \frac{1}{x^2} + 1$

with:

1	$y = x + 1$	3	$y = \sqrt{x^2} + 1$
2	$y = x^2 + 1$	4	$y = \frac{1}{x^2} + 1$

In the examples with `tabulate` we honor the original dimensions but you can also set the `width`, combined with a `distance`. Instead of a dimension the `width` parameter can be set to `fit`.

In case one wonders, of	in tables) that can be	lems but when used in
course regular columns	swapped. For as far as	situations where one
can be used, but this is	possible footnotes are	knows what goes in, it
an alternative that actu-	supported but of course	is quite powerful any-
ally gives you balancing	floats are not.	way. It also has a rela-
for free, but of course	So, this rows based	tively simple implemen-
with the limitation that	mechanism is not the	tation.
we have lines (or cells	solution for all prob-	

In the previous rendering we have set the `width` as mentioned but also set `align` to `verytolerant`, `stretch` so that we don't overflow lines. The `before` and `after` parameters are set to `\blank`.

Rows becoming columns

23.2 Colofon

Author	Hans Hagen & Mikael Sundqvist
ConT _E Xt	2025.02.12 08:44
LuaMetaT _E X	2.11.06 20250218
Support	www.pragma-ade.com contextgarden.net ntg-context@ntg.nl