# BachoTEX

May 1, 2024

Marek Ryćko

# Arithmetic expressions and their evaluation in Plain TEX

# Playing with programming in Plain TEX

# An exercise in programming in Plain TEX

# Exploring the boundaries of programming in Plain TEX

**notation of expressions**

in mainstream
programming languages

**Algol 60**

my first serious programming language

```
integer result;
result := 2 + (33 - 7);
```

almost 30 years later...

**Python**

my current favorite language

```
result = 2 + (33 - 7)
```

in both cases:

**infix notation**

```
integer result;
result := 2 + (33 - 7);

result = 2 + (33 - 7)
```

also used:

**prefix notation**

like function calls

# Python

```
# in Python:

import operator as op
result = op.add(2, op.sub(33, 7))
```

in some programming languages:

**postfix notation**

postfix notation

**Reverse Polish Notation (RPN)**
reverse Łukasiewicz notation

postfix notation

Reverse Polish Notation (RPN)
**reverse Łukasiewicz notation**

from our typography area:

**Postscript**

```
/result 33 7 sub 2 add def
```

```
/result 33 7 sub 2 add def
/result 26 2 add def
/result 28 def
% result == 28
```

no brackets needed

less readable

my favorite programming language specialized to typography:

TEX

```
% Python:
% result = 3 + 17

% TeX:
\newcount\result
\result=3
\advance\result 17
```

```
% Python:
% result = 3 + 17

% TeX:
\newcount\result
\result=3
\advance\result by 17
```

```
% Python: result = (3 + 17) - (9 - 1)

\newcount\result
\newcount\partial
\partial=9
\advance\partial -1
\result=3
\advance\result 17
\advance\result -\partial
```

# Data types in TEX

**in TEX**

no clear notion of data type

hint:

TEX's registers

```
\newcount
\newdimen
\newskip
\newmuskip
```

the `\newcount` command reserves (declares) a variable of the `count` type

`count` = integer

the `\newdimen` command reserves (declares) a variable of the `dimen` type

`dimen` = length, dimension

the `\newskip` command
reserves (declares) a variable
of the `skip` type

`skip` = glue, streachable and shrinkable
length, dimension

the `\newmuskip` command
reserves (declares) a variable
of the `muskip` type

`muskip` = math glue,
streachable and shrinkable length,
dimension for the math mode

# simple types

```
count
dimen
skip
muskip
```

**simple types**

each element has
a fixed number of components

# simple types

`count` – a single number
`dimen` – a (real) number with unit
`skip` – `dimen` plus 2 objects
`muskip` – similar to `dimen`

there are some more simple types in TEX

there are some more
simple types in TEX

without registers

the `real` type

called also float or decimal

in TEX's grammar:

⟨decimal constant⟩

2.33
3.1415926
2024

the `bool` type

called also Boolean

just two values

`true` and `false`

In TEX the `bool` type...

In TEX the `bool` type…
just does not exist

there are some traces
of the type

```
\iftrue
    This is wise.
\else
    This is stupid.
\fi
```

the elements of the `bool` type are hidden, internal

# Arithmetic calculations in TEX

for the “registered” simple types

`count`, `dimen`, `skip`, `muskip`

expressing the calculations to be done is complicated

```
% result = (3 + 17) - (9 - 1)

\newcount\result
\newcount\partial
\partial=9
\advance\partial -1
\result=3
\advance\result 17
\advance\result -\partial
```

for the `real` type:

there is an operation
of multiplying
a `real` value
by one of the *length* values

```
\hsize=16cm
\vsize=1.4142\hsize
```

but it is not possible
to calculate the `real` value
and prepare it beforehand

it is not possible
even in TEX's notation

```
\newreal\temp
\temp=3.14
\advance\temp by 1.41

% the result would be 4.55
```

no operations
on `bool` values

no `\and`, `\or`,
`\not` operations

# The summary of arithmetical operations in TEX

# 1.

there are six simple data types

`count`, `dimen`, `skip`, `muskip`
`real`, `bool`

# 2.

TEX has registers only for four of the types

`count`, `dimen`, `skip`, `muskip`

**3.**

The **notation** for calculations
on the values of
first four data types

**is very inconvinient**

**4.**

The only built-in operation
involving the `real` type is

multiplication by one of the *length* values

**5.**

There are no operations on the `bool` values

# The challenge

The possibility of using
**arithmetical expressions**

of the **six simple types**

programmed with

the commands of classic TEX
and macros of **Plain TEX**

only

to make it usable
in **any version of TEX**

The possibility to build

**complex, hierarchical expressions**

possibly with

nested parentheses/braces

possibly with

(nested (parentheses/braces))

possibly with

{nested {parentheses/braces}}

and a **uniform** way
to **assign** values
to variables

# Design decisions

Design decisions:

# The notation of expressions

following the **nature of TEX**

The very basic TEX's notation
for user-defined action is

**a macro name with some parameters**

```
\doit{one}{another}{last}
```

**prefix notation**
like function call

so for **adding** some values
in TEX

the natural notation would be

\+{3}{17}

the natural way
of writting **subexpressions**

with the use of nested parentheses
in TEX: **nested braces**

```
% (3 + 17) - (9 - 1)

\-{\+{3}{17}}{\-{9}{1}}
```

**No.** The TEX's version

is **not** more readable
than the Python's version

But it **is** more readable than...

```
% (3 + 17) - (9 - 1)

\newcount\result
\newcount\partial
\partial=9
\advance\partial -1
\result=3
\advance\result 17
\advance\result -\partial
```

Design decisions:

# Context dependence of operation symbols

context dependent meaning
of operation symbols like

\+ \- * \/

**1.**

the meaning is **local**
within the expression

**2.**

the meaning depends
on the resulting type

in a typical programming language

5 + 8

```
5 [integer +] 8
```

5 + 8.3

```
to_real(5) + 8.3
```

5.0 + 8.3

```
5.0 [real +] 8.3
```

in TEX's `eval` module
by design decision

**no guessing** of the resulting type

the resulting **type**
of each expression and subexpression

must be **given explicitly**

so the type-dependent meaning of each operation symbol

```
\+ \- \* \/
\div \mod \and \or
```

is known in advance

Design decisions:

**Variables**

# how to store
# the values of the six types

the usual TEX's way

**registers**

the usual TEX's way

**registers?**

let’s list the
**disadvantages** of registers

**1.**

registers are of
only for four simple types

**1.**

registers are of
only for four simple types

`count dimen skip muskip`

**even if** we decided to store the variables
of those four types
in **registers**

**2.**

there are only 256 registers
of each type

**2.**

there may be **not enough variables**
for bigger macro sets

**3.**

the way of storing values
would **not** be **uniform**

yes, I do like

**uniformity**

and I do like

**space**

and I do like

**practically unlimited space**
for variables

so the decision is to
keep values as

**the macros**

so the decision is to
keep values as

**the macro bodies**

the results of
evaluating the expressions

are like the results of
**defining macros**

```
\def\somecount  {17}
\def\somedimen  {1234sp}
\def\someskip   {22mm plus 1pt minus 1fil}
\def\somemuskip {33dd plus 2mm}
\def\somereal   {3.14}
\def\somebool   {\true}
```

the apparent disadvantages

of using **macros**
as variables

**1.**

more **memory** taken
as compared to registers

**2.**

more **time** used
as compared to registers

but

**but**

thankfully TEX's implementation
was completed about 42 years ago

since then

the computers’ memory size increased about **2 milion times**

since then

the computers’ speed
increased about **2 milion times**

we can select
**ease of use**

over the **speed and size** efficiency

so the decision was to
create uniform
**evaluation and assignment commands**

that
take **three groups of parameters**

**1.**

the arithmetic expression

**2.**

the result type

**3.**

the variable name

all data in a single command

compute ⟨expression⟩
of the type ⟨type⟩
and assign the result to ⟨variable name⟩

Design decisions:

# The notation of the evaluate-and-assign command

following the spirit of TEX
in the area of

**assigning a value to a macro name**

```
\def\macro{micro}
```

```
\def\macro{micro}
\edef\expanded{\macro}
```

```
\def\macro{micro}
\edef\expanded{\macro}
\def\expanded{micro}
```

`\edef`

do something with the parameter
before assigning the value
to the macro name

`\edef\macro{...}`

do something with the parameter
before assigning the value
to the macro name

following the same spirit...

```
\defc\sum{\+{3}{9}}
```

```
\defc\sum{\+{3}{9}}

c         - result type (here: count)
\sum      - variable name
\+{3}{9} - expression to be computed
```

```
\defc\sum{\+{3}{9}}

\def\sum{12}
```

there are similar commands
for all the six types

\defc
\defd
\defs
\defm
\defr
\defb

```
\defc - count
\defd - dimen
\defs - skip
\defm - muskip
\defr - real
\defb - bool
```

# Minimal requirements

# 1.

category codes

# 2.

\outer symbols

# 1.

category codes

the **underscore** character _
is assigned the category “letter”

for the time of **reading the module**

the control sequences like:

`\eval_local_notation_muskip`

are used internally

after reading the module
**the previous category** of
underscore characters _
**is restored**

no conflict
with the user’s habits

# 2.

\outer symbols

the problem in Plain TEX:

```
\outer\def\+{\tabalign}
```

the control sequence

cannot be used
in a macro definition or parameter

```
\outer\def\+{\tabalign}

\input eval
% includes \def\+{\tabalign}
```

in the unlikely case
of other meaning of

the original meaning is saved
and can be restored by user

you cannot use

`\outer\+`

and the module `eval`

# Implementation issues

the implementation of

the operations
of the type `real`

no explicit `real` operations
in TEX

but the `real` numbers
are used

```
\newdimen\somedimen
\somedimen = 17.5pt
```

17.5pt

17.5mm

17.5in

17.5sp

no `real` operations
in TEX

in the `eval` module

**three different techniques**

to perform
the `real` operations

# 1.

addition and subtraction

`\+` and `\-`

**step A.**

attach the `pt` suffixes
to both arguments

**step B.**

add the resulting dimensions

```
% 3.14 + 1.41

\newdimen\temp
\temp=3.14pt
\advance\temp by 1.41pt
```

**another step B.**

or subtract the resulting dimensions

```
% 3.14 - 1.41

\newdimen\temp
\temp=3.14pt
\advance\temp by -1.41pt
```

**step C.**

remove the `pt`
from the result

**No.**

It’s not so simple.

```
\newdimen\temp
\temp=3.14pt
% 4 characters of category ‘other’
% 2 characters of category ‘letter’

\edef\macro{\the\temp}
% in the \macro we have
% 6 characters of category ‘other’
```

we have to chop off
the `pt` suffix

but normally the `pt` are letters
and in the macro result
we have the `pt` of different “color”

we have to chop off
the `pt` suffix

but normally the `pt` are letters
and in the macro result
we have the `pt` of different “color”

so even the category codes
are some obstacle
to the `real` addition and subtraction

but an obstacle
is not a problem

**2.**

multiplication of `real` numbers

`\*`

# TEX does multiplication...

multiply:

```
1.41 * 1.42
```

```
% 1.41 * 1.42

\newdimen\length
\length = 1.42pt
\result = 1.41\length
```

```
\newdimen\length
\length = 1.42pt
\result = 1.41\length

\edef\macroresult{\the\result}
```

```
\newdimen\length
\length = 1.42pt
\result = 1.41\length

\edef\macroresult{\the\result}
\def\macroresult{2.0022pt}
```

**3.**

division of `real` numbers

`\/`

the algorithm implemented in TEX
by Thomas Rokicki
March 29, 1989

for the `epsf` macro set

calculating the proportion

$$y = rx/t$$

for given $r$, $x$, $t$

make $y$ such that:

$$y/r = x/t$$

for given $r$, $x$, $t$

all the four values

$x, y, r, t$

are lengths or dimensions

for $x$ = `1pt`
we have the division of real values
with the `pt` attached

calculating the proportion

$$y = 1\mathrm{pt} \times x/t$$

for given $x$, $t$

# Examples

**1.**

exponentiation

**1.**

there is no exponentiation

but...

```
\*{2}{\*{2}{\*{2}{\*{2}{2}}}}

% the result is 32
```

```
\*{7}{\*{7}{\*{7}{\*{7}{7}}}}

% the result is 16807
```

# 2.

## `\div` and `\mod` operations

```
\defc\di{\div {12345} {100}}
% result: 123

\defc\mo{\mod {12345} {100}}
% result: 45
```

```
\defc\div{\div {12345} {100}}
% result: 123

\defc\mod{\mod {12345} {100}}
% result: 45
```

**3.**

notation of `count` values (integers)

```
\defc\oct{'100}
```

```
\defc\oct{'100}

\def\oct{64}
```

```
\defc\oct{'100}

\def\oct{64}
\defc\oct{64}
```

```
\defc\hex{"FF}
```

```
\defc\hex{"FF}

\def\hex{255}
\defc\hex{255}
```

```
\defc\ord{‘!}
```

```
\defc\ord{‘!}

\def\ord{33}
\defc\ord{33}
```

# 4.

type conversion

```
\defc\strange
    {%
    \+2{\+{\*{17}{\+{\prevgraf}{1}}}%
    {\-{\vsize}{\hsize}}}%
    }%
```

```
\defc\strange
    {%
    \+2{\+{\*{17}{\+{\prevgraf}{1}}}%
    {\-{\vsize}{\hsize}}}%
    }%

\def\strange{11367078}
```

```
\defc\strange
    {%
    \+2{\+{\*{17}{\+{\prevgraf}{1}}}%
    {\-{\vsize}{\hsize}}}%
    }%

\def\strange{11367078}
\defc\strange{11367078}
```

# 5.

muliplying
`real` by `dimen`

\hsize16cm

```
\hsize16cm
% \hsize is 455.24408pt
```

```
\hsize16cm
% \hsize is 455.24408pt

\defd\pagew{210mm}
% equivalent of \def\pagew{597.50787pt}
```

```
\defd\hmargin{\*{0.5}{\-\pagew\hsize}}
```

this is the one-liner
I really needed

# History

I designed this module
and programmed the first version
in **1993**

The `\def`-like notation was created in **1999**

I’ve been using this module
for **more then 30 years**

Still there is a lot
that could be done
or done better