UNIVERSITÀ DEGLI STUDI DI PADOVA

The Typst language

Advanced Topics in Programming languages presentation

Alberto Lazari – 2089120

September 14, 2023

Markup languages

Procedural markup

- Basic built-in commands for simple actions
- Macros for complex procedures

Procedural markup

- Basic built-in commands for simple actions
- Macros for complex procedures

Procedural markup

- Basic built-in commands for simple actions
- Macros for complex procedures

Troff

- Early typesetting system
- Imperative and strictly procedural

```
.ce
This is a single centered
line
.LP
.ce 3
followed by
a sequence of three (3)
centred lines
```

TEX

- Smart line breaks
- Advanced layout algorithms

```
\magnification=\magstep1
\baselineskip=12pt
\hsize=6.3truein
\vsize=8.7truein
\font\footbf=cmbx10 at 8truept

\font\bigrm=cmr12 at 14pt
\centerline{\bigrm The title}

\bigskip\bigskip
\centerline{\bf Abstract}
\smallskip
{\narrower\noindent
The abstract.\par}

\bigskip
\beginsection 1. Introduction.

This is the start of the introduction.
\bye
```

TeX

- Smart line breaks
- Advanced layout algorithms
- Still procedural

```
\magnification=\magstep1
\baselineskip=12pt
\hsize=6.3truein
\vsize=8.7truein
\font\footbf=cmbx10 at 8truept

\font\bigrm=cmr12 at 14pt
\centerline{\bigrm The title}

\bigskip\bigskip
\centerline{\bf Abstract}
\smallskip
{\narrower\noindent
The abstract.\par}

\bigskip
\beginsection 1. Introduction.

This is the start of the introduction.
\bye
```

Descriptive markup

- Structure rather than appearance
- Same structure, different styling
- Reusability
- Less boilerplate

Descriptive markup

- Structure rather than appearance
- Same structure, different styling
- Reusability
- Less boilerplate

Descriptive markup

- Structure rather than appearance
- Same structure, different styling
- Reusability
- Less boilerplate

Descriptive markup

- Structure rather than appearance
- Same structure, different styling
- Reusability
- Less boilerplate

Descriptive markup

- Structure rather than appearance
- Same structure, different styling
- Reusability
- Less boilerplate

LaTeX

- Set of useful TeX macros
- *Describe content* vs. *describe output*
- Document class for the style
- Tedious debugging

```
\documentclass{article}
\begin{document}

\section{Introduction}
This is a simple example

\begin{itemize}
  \item First item
  \item Second item
\end{itemize}

\end{document}
```

LaTeX

- Set of useful TeX macros
- *Describe content* vs. *describe output*
- Document class for the style
- Tedious debugging

```latex
\documentclass{article}
\begin{document}

\section{Introduction}
This is a simple example

\begin{itemize}
  \item First item
  \item Second item
\end{itemize}

\end{document}
```

LaTeX

- Set of useful TeX macros
- *Describe content* vs. *describe output*
- Document class for the style
- Tedious debugging

```
\documentclass{article}
\begin{document}

\section{Introduction}
This is a simple example

\begin{itemize}
  \item First item
  \item Second item
\end{itemize}

\end{document}
```

LaTeX

- Set of useful TeX macros
- *Describe content* vs. *describe output*
- Document class for the style
- Tedious debugging

```latex
\documentclass{article}
\begin{document}

\section{Introduction}
This is a simple example

\begin{itemize}
  \item First item
  \item Second item
\end{itemize}

\end{document}
```

Markdown

- Lightweight syntax for more powerful language (HTML)
- Intuitive
- Limited

```
# Markdown
Text can be *emphasized* or
**strong**.
Here is a [link](https://
github.com)

Plain text is:
- Simple to write
- Easy to read
```

Markdown

- Lightweight syntax for more powerful language (HTML)
- Intuitive
- Limited

```
# Markdown
Text can be *emphasized* or
**strong**.
Here is a [link](https://
github.com)

Plain text is:
- Simple to write
- Easy to read
```

Markdown

- Lightweight syntax for more powerful language (HTML)
- Intuitive
- Limited

```
# Markdown
Text can be *emphasized* or
**strong**.
Here is a [link](https://
github.com)

Plain text is:
- Simple to write
- Easy to read
```

Typst

The Typst language

- Open source typesetting system
- Lightweight syntax
- Functional programming language
- Fast compile times for instant preview

The Typst language

- Open source typesetting system
- Lightweight syntax
- Functional programming language
- Fast compile times for instant preview

The Typst language

- Open source typesetting system
- Lightweight syntax
- Functional programming language
- Fast compile times for instant preview

The Typst language

- Open source typesetting system
- Lightweight syntax
- Functional programming language
- Fast compile times for instant preview

Syntax modes

- Markup: `[ content ]`
- Math: `$ math $`
- Code: `{ code }`

Syntax modes

Markup mode

```
= Example

Some _text_ in *markup*
```

Example

Some *text* in **markup**

Syntax modes

Markup mode

```
= Example

Some _text_ in *markup*
```

Example

Some *text* in **markup**

Math mode

```
If $n in NN$, then:
$ sum_(i = 0)^n k x_i $
```

If $n \in \mathbb{N}$, then:

$$\sum_{i=0}^{n} k x_i$$

Syntax modes

Markup mode

```
= Example

Some _text_ in *markup*
```

Example

Some *text* in **markup**

Math mode

```
If $n in NN$, then:
$ sum_(i = 0)^n k x_i $
```

If $n \in \mathbb{N}$, then:

$$\sum_{i=0}^{n} k x_i$$

Code mode

```
#{
  let f = x => y => x + y
  f(1)(2)
}
```

```
3
```

Markup mode

Markup mode

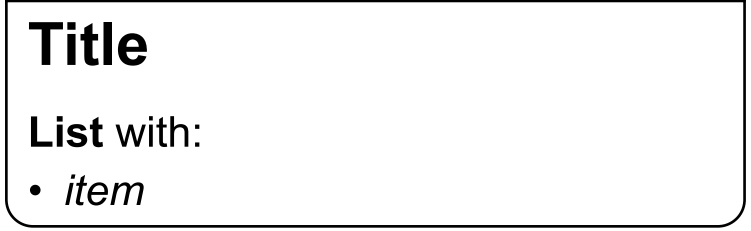
- Default syntax mode

Markup mode

- Default syntax mode
- Syntactic sugar for function calls

```
= Title

*List* with:
- _item_
```



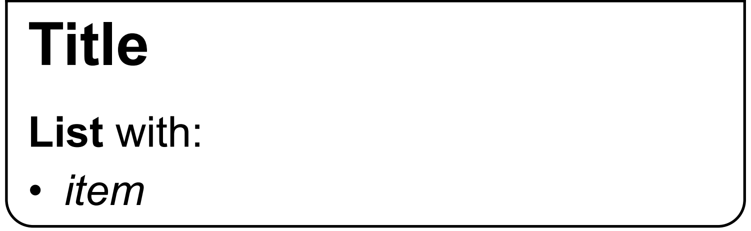
Title

List with:

- *item*

$\Longleftrightarrow$

```
#{
  heading("Title"); parbreak()
  text(strong("List") + " with:")
  list(emph("item"))
}
```



Title

List with:

- *item*

Content type

- Tree of content elements
- From functions or markup
- Document as join of all returned contents

```
#let content = [_example_ *text*]
This is the content: "#content" \
Representation: #repr(content)
```

This is the content: “*example* **text**”
Representation: sequence(
  children: (emph(body: [example]), [ ],
strong(body: [text])),
)

Content type

- Tree of content elements
- From functions or markup
- Document as join of all returned contents

```
#let content = [_example_ *text*]
This is the content: "#content" \
Representation: #repr(content)
```

This is the content: “*example* **text**”
Representation: sequence(
  children: (emph(body: [example]), [ ],
strong(body: [text])),
)

Content type

- Tree of content elements
- From functions or markup
- Document as join of all returned contents

```
#let content = [_example_ *text*]
This is the content: "#content" \
Representation: #repr(content)
```

This is the content: “*example* **text**”
Representation: sequence(
  children: (emph(body: [example]), [ ],
strong(body: [text])),
)

Problem

1. Everything is a function call
2. Functions are expressions

Problem

1. Everything is a function call
2. Functions are expressions

$\Longrightarrow$ Final document needs to be a series of `content` values

Problem

1. Everything is a function call
2. Functions are expressions

⟹ Final document needs to be a series of `content` values

Solution: explicitly join every single one?

```
#let document = {
  emph("This ") + [is a test to ] + text(fill: red, "join ") + [eveything]
}
#document
```

This is a test to join eveything

Joining

- Every line returns a value (or `none`)
- A block returns the join of every generated value

```
#let document = {
  emph("This ")
  [is a test to ]
  text(fill: red, "join ")
  [eveything]
}
#document
```

This is a test to join eveything

Joining

- Every line returns a value (or `none`)
- A block returns the join of every generated value
- Conditionals and loops are expressions too

```
#for x in (1, 2, 3) [
  - #x #if x == 1 [ (first) ]
]
```

- 1 (first)
- 2
- 3

Code mode

Type system

- Dynamic typing
- Few implicit conversions (`string` → `content`)
- No custom types
- No subtyping

Types

- `content` $\simeq \top$

Types

- `content` $\simeq \top$ ($\neq$ `any`)

Types

- `content` $\simeq \top$ ($\neq$ `any`)
- `none` $= \bot$

Types

- `content` $\simeq \top$ ($\neq$ `any`)
- `none` $= \bot$
- programming (`integer`, `boolean`, `string`, `function`, …)

Types

- `content` $\simeq \top$ ($\neq$ `any`)
- `none` $= \bot$
- programming (`integer`, `boolean`, `string`, `function`, …)
- data structures (`array`, `dictionary`)

Types

- `content` $\simeq \top$ ($\neq$ `any`)
- `none` $= \bot$
- programming (`integer`, `boolean`, `string`, `function`, …)
- data structures (`array`, `dictionary`)
- styling (`length`, `angle`, `color`, …)

Unique copies

No *reference* types, only *value* types

```
#let array = (1, 2, 3)
#let copy = array
#copy.push(4)
Array = #array \
Copy = #copy
```

Array = `(1, 2, 3)`
Copy = `(1, 2, 3, 4)`

Functions

- First class values

Functions

- First class values
- Closures

Functions

- First class values
- Closures
- Pure (user-defined)

Functions – examples

Closure

```
#{
  let var = 1
  let f(x) = { x + var }
  var = 10
  f(1)
}
```

```
2
```

Functions – examples

Closure

```
#{
  let var = 1
  let f(x) = { x + var }
  var = 10
  f(1)
}
```

```
2
```

Pure

```
#{
  let var = 1
  let g() = { var += 1 }
  g()
}
```

```
error: variables from outside the
function are read-only and cannot
be modified
```

Functions – examples

Closure

```
#{
  let var = 1
  let f(x) = { x + var }
  var = 10
  f(1)
}
```

```
2
```

Pure

```
#{
  let var = 1
  let g() = { var += 1 }
  g()
}
```

```
error: variables from outside the
function are read-only and cannot
be modified
```

First class value

```
#{
  let curried-map = f => (..l) => {
    l.pos().map(f)
  }
  curried-map(x => x + 1)(0, 1, 2)
}
```

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

Functions – recursive let binding

```
#{
  let map(f, ..items) = {
    let list = items.pos()
    if list.len() == 0 { return list }

    let (x, ..rest) = list
    ( f(x), ..map(f, ..rest) )
  }
  map(x => x + 1, 0, 1, 2)
}
```

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

Functions – recursive let binding

```
#{
  let map(f, ..items) = {
    let list = items.pos()
    if list.len() == 0 { return list }

    let (x, ..rest) = list
    ( f(x), ..map(f, ..rest) )
  }
  map(x => x + 1, 0, 1, 2)
}
```

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

```
#{
  let map = f => (..items) => {
    let list = items.pos()
    if list.len() == 0 { return list }

    let (x, ..rest) = list
    ( f(x), ..map(f)(..rest) )
  }
  map(x => x + 1)(0, 1, 2)
}
```

```
error: unknown variable: map
```

Functions – recursive let binding

```
#{
  let map(f, ..items) = {
    let list = items.pos()
    if list.len() == 0 { return list }

    let (x, ..rest) = list
    ( f(x), ..map(f, ..rest) )
  }
  map(x => x + 1, 0, 1, 2)
}
```

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

```
#{
  let map = {
    let rec = map => f => (..items) => {
      let list = items.pos()
      if list.len() == 0 { return list }

      let (x, ..rest) = list
      ( f(x), ..map(map)(f)(..rest) )
    }
    rec(rec)
  }
  map(x => x + 1)(0, 1, 2)
}
```

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

Parameters

- Positional: `#f(x, y)`

Parameters

- Positional: `#f(x, y)`
- Currying (not idiomatic): `#g(x)(y)`

Parameters

- Positional: `#f(x, y)`
- Currying (not idiomatic): `#g(x)(y)`
- Variadic: `#let h(..args) = { ... }`

Parameters

- Positional: `#f(x, y)`
- Currying (not idiomatic): `#g(x)(y)`
- Variadic: `#let h(..args) = { ... }`
- Named: `#text("hello", color: red)`

Named parameters

Typst

```
#text(color: red, "text")
// Order-independent
#text("text", color: red)
// Optional
#text("text")
```

Named parameters

Typst

```
#text(color: red, "text")
// Order-independent
#text("text", color: red)
// Optional
#text("text")
```

LaTeX

```
\inputminted[lineos, bgcolor=gray]{rust}{ex.rs}
% Order-independent
\inputminted[bgcolor=gray, lineos]{rust}{ex.rs}
% Optional
\inputminted{rust}{ex.rs}
```

LaTeX – optional parameters

```
\newcommand{\mysum}[2][n]{
  \sum_{i = 0}^#1 #2
}
$$
  \mysum{x_i}
  \mysum[\infty]{x_i}
$$
```

$\Longrightarrow$

```
#let mysum(exp, limit: $n$) = {
  $sum_(i = 0)^limit exp$
}
$
  #mysum($x_i$)
  #mysum($x_i$, limit: $infinity$)
$
```

$$\sum_{i=0}^{n} x_i \sum_{i=0}^{\infty} x_i$$

LaTeX – multiple optional parameters

```
% Missing { inserted.
\newcommand{\mysum}[3][i][n]{
  \sum_{#1 = 0}^#2 #3
}
$$
  \mysum{x_i}
  \mysum[j][\infty]{x_j}
$$
```

$\Longrightarrow$

```
#let mysum(exp, index: $i$, limit: $n$) = {
  $sum_(index = 0)^limit exp$
}
$
  #mysum($x_i$)
  #mysum($x_j$, index: $j$, limit: $infinity$)
$
```

$$\sum_{i=0}^{n} x_i \sum_{j=0}^{\infty} x_j$$

Partial application

```
#{
  let mysum(exp, index: $i$, limit: $n$) = $sum_(index = 0)^limit exp$

  mysum = mysum.with(limit: $infinity$)
  $ #mysum($x_i$) $

  let mysum = mysum.with(limit: $4$, index: $x$)
  $ #mysum($x$) = 0 + 1 + ... + 4 = 10 $
}
```

$$\sum_{i=0}^{\infty} x_i$$

$$\sum_{x=0}^{4} x = 0 + 1 + \ldots + 4 = 10$$

Compiler

Compiler

Steps to compile source files to PDF:

1. Parsing
2. Evaluation
3. Lifting
4. Layout
5. Export

Steps to compile source files to PDF:

1. Parsing
2. Evaluation
3. Lifting
4. Layout
5. Export

Evaluation

No syntax errors $\implies$ evaluation can happen:

1. Joined content value
2. Top level bindings

Evaluation

No syntax errors $\Longrightarrow$ evaluation can happen:

1. Joined content value
2. Top level bindings

Evaluation

No syntax errors $\Longrightarrow$ evaluation can happen:

1. Joined content value
2. Top level bindings

Evaluation – markup

- Markup nodes → `content`
- Code blocks evaluated to final value (joined) → `content`
- Everything joined in the process

Evaluation – markup

- Markup nodes → `content`
- Code blocks evaluated to final value (joined) → `content`
- Everything joined in the process

Evaluation – markup

- Markup nodes → `content`
- Code blocks evaluated to final value (joined) → `content`
- Everything joined in the process

Evaluation – bindings

When evaluating a `#let` binding:

- Store `(name, value)` in the scope
- Return `none`

Evaluation – bindings

When evaluating a `#let` binding:

- Store `(name, value)` in the scope (closures are values)
- Return `none`

Evaluation – bindings

Closures can't be statically checked (only syntax)

```
#let f() = {
  let g(x)(y) = { x + y }
}
```

```
error: expected equals sign
   ┌─ /sections/compiler.typ:67:16
   │
67 │         let g(x)(y) = { x + y }
   │                 ^
```

Evaluation – bindings

Closures can't be statically checked (only syntax)

```
#let x = 0
#let val() = {
  x += 1
}
This compiles fine
```

This compiles fine

```
#let x = 0
#let val() = {
  x += 1
}
#val()
```

```
error: variables from outside the function
are read-only and cannot be modified
```

Evaluation – bindings

Closures can't be statically checked (only syntax)

```
#let x = 0
#let val() = {
  x += 1
}
#val()
```

```
error: variables from outside the function
are read-only and cannot be modified
```

```
#let x = 0
#let val = {
  x += 1
}
#x
```

```
1
```

Modules

- Evaluation of a single source file: `(content, bindings)`
- `#include "module.typ"` → content
- `#import  "module.typ"` → bindings

Modules

- Evaluation of a single source file: `(content, bindings)`
- `#include "module.typ"` → content
- `#import  "module.typ"` → bindings
- Immutability

Modules

- Evaluation of a single source file: `(content, bindings)`
- `#include "module.typ"` → content
- `#import  "module.typ"` → bindings
- Immutability
- Caching

Improvements

Syntax

- LaTeX inconsistent syntax for implementation reasons: `\command{...}` vs `\begin{command} ... \end{command}`
- TeX can alter and create syntax: `$x + y\]`
- Typst has a well-defined syntax

Syntax

- LaTeX inconsistent syntax for implementation reasons: `\command{...}` vs `\begin{command} ... \end{command}`
- TeX can alter and create syntax: `$x + y\]`
- Typst has a well-defined syntax

Syntax

- LaTeX inconsistent syntax for implementation reasons: `\command{...}` vs `\begin{command} ... \end{command}`
- TeX can alter and create syntax: `$x + y\]`
- Typst has a well-defined syntax

Macros

- Simpler and immediate → more intuitive
- No scoping → side effects and package conflicts

Error reporting

- $\TeX$'s interactive error correction
- Well-defined syntax
- Type system

Error reporting – interactive correction

LaTeX

```
$x + y
Missing $ inserted.
```

Typst

```
$x + y
      ^
expected closing dollar sign
```

Error reporting – syntax

LaTeX

```
\section
Missing \endcsname inserted.
Missing \endcsname inserted.
Missing \endcsname inserted.
...
```

Typst

```
#heading()
```

```
error: missing argument: body
    ┌─ /sections/improvements.typ:94:16
    │
94  │          #heading()
    │                  ^^
```

Error reporting – type system

LaTeX

```
\baselineskip=normal
Missing number, treated as zero.
Illegal unit of measure (pt inserted).
```

Typst

```
#set par(leading: "normal")
```

```
error: expected length, found string
   ┌─ /sections/improvements.typ:70:24
   │
70 │       #set par(leading: "normal")
   │                         ^^^^^^^^
```

Computational foundations

No data structures in $\TeX \Rightarrow$ provides a package for everything

$\LaTeX$

```latex
\usepackage{trimspaces}
\trim@post@space{Text  }
```

Typst

```typst
#{ "Text  ".trim(at: end) }
```

Computational foundations

LaTeX

```latex
\usepackage{listofitems}
\def\tabelize#1{
  \readlist\animals{#1}
  \begin{table}
    \textbf{Animal} \\
    \foreachitem\a\in\animals{
      \a \\
    }
  \end{table}
}
```

Typst

```typst
#let tabelize(str) = {
  let animals = str.split(", ")
  table([*Animal*], ..animals)
}
#tabelize("Tiger, Giraffe, Cougar")
```

Computational foundations

LaTeX

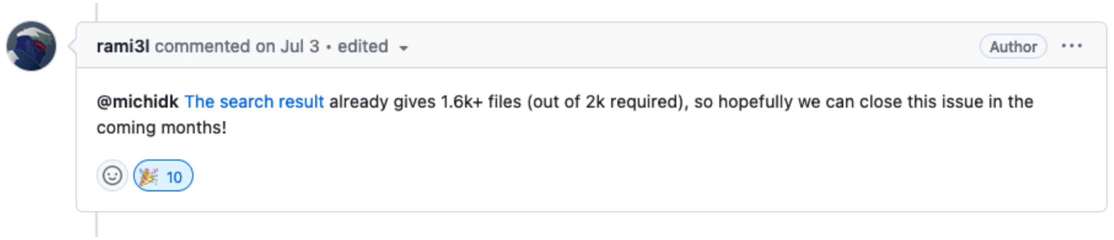
```
\newcount\i \i=0
\loop
  \advance \i by 1
  Variable i = \the\i
\ifnum \i<5 \repeat
```

Typst

```
#let i = 0
#while i < 5 {
  i += 1
  [Variable i = #i]
}
```

GitHub support

- Pull request to add Typst support in GitHub repos
- Needs more popularity



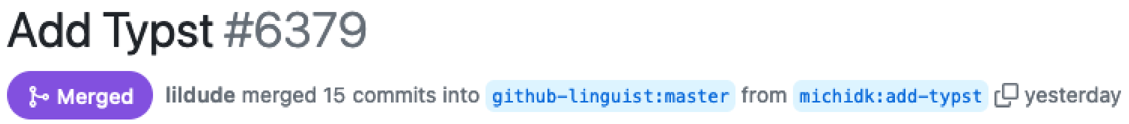
rami3l commented on Jul 3 • edited

Author

@michidk The search result already gives 1.6k+ files (out of 2k required), so hopefully we can close this issue in the coming months!

🎉 10

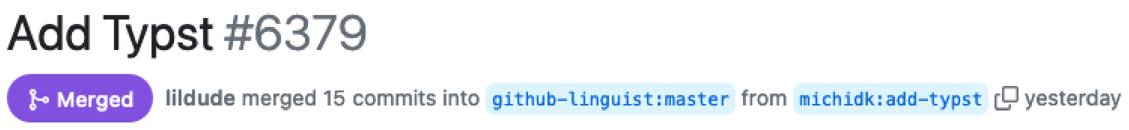
Popularity



Add Typst #6379

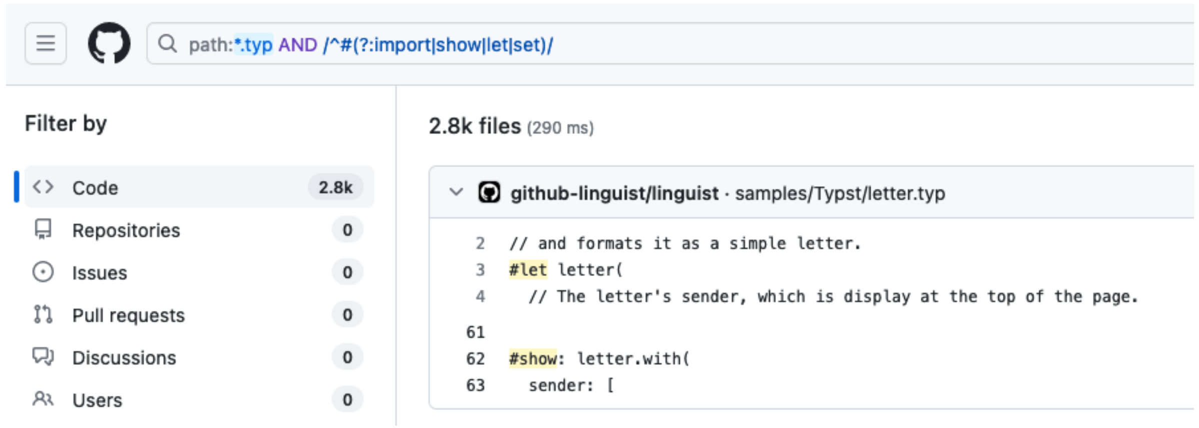
Merged **lildude** merged 15 commits into `github-linguist:master` from `michidk:add-typst` yesterday

Popularity



Add Typst #6379

Merged lildude merged 15 commits into github-linguist:master from michidk:add-typst yesterday



path:*.typ AND /^#(?:import|show|let|set)/

Filter by

Code	2.8k
Repositories	0
Issues	0
Pull requests	0
Discussions	0
Users	0

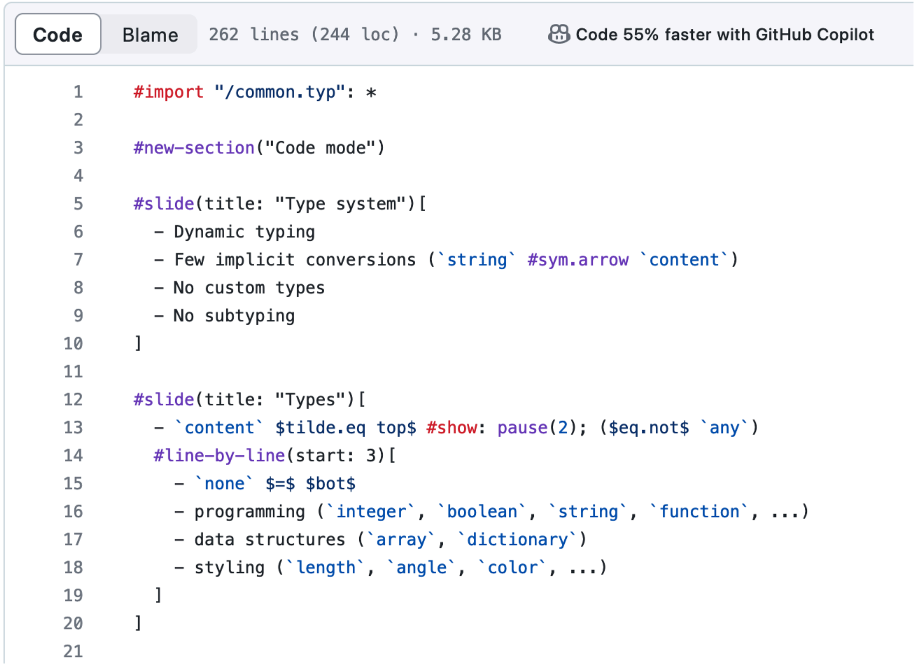
2.8k files (290 ms)

github-linguist/linguist · samples/Typst/letter.typ

```
// and formats it as a simple letter.
#let letter(
  // The letter's sender, which is display at the top of the page.

#show: letter.with(
  sender: [
```

Latest news



Code Blame 262 lines (244 loc) · 5.28 KB Code 55% faster with GitHub Copilot

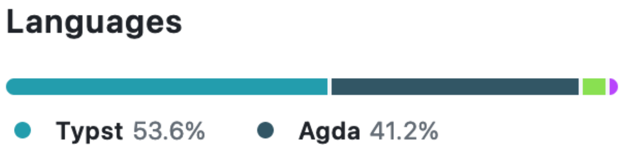
```
#import "/common.typ": *

#new-section("Code mode")

#slide(title: "Type system")[
  - Dynamic typing
  - Few implicit conversions (`string` #sym.arrow `content`)
  - No custom types
  - No subtyping
]

#slide(title: "Types")[
  - `content` $tilde.eq top$ #show: pause(2); ($eq.not$ `any`)
  #line-by-line(start: 3)[
    - `none` $=$ $bot$
    - programming (`integer`, `boolean`, `string`, `function`, ...)
    - data structures (`array`, `dictionary`)
    - styling (`length`, `angle`, `color`, ...)
  ]
]

```



Languages

● **Typst** 53.6% ● **Agda** 41.2%

Sources

- **Laurenz Mädje (typst co-creator) Master's thesis**: https://www.user.tu-berlin.de/laurmaedje/programmable-markup-language-for-typesetting.pdf
- **Typst official documentation**: https://typst.app/docs/

Thanks for the attention