Plum

Plum lets you create UML class diagrams in Typst; inspired by but *not* compatible with PlantUML.

v0.0.1

https://github.com/SillyFreak/typst-plum

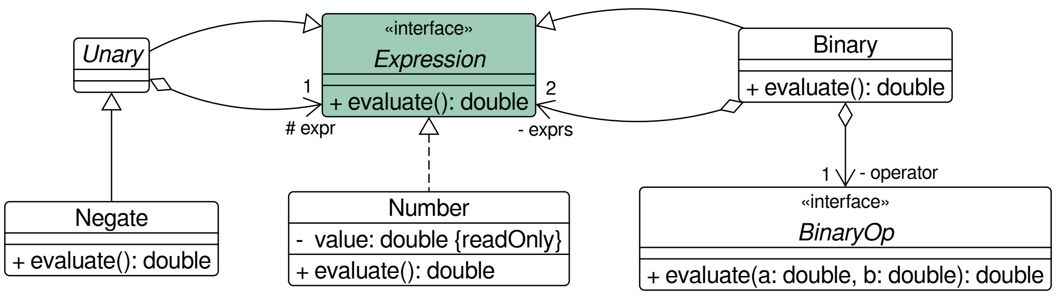
Clemens Koza

Contents

I Introduction

Plum lets you create UML class diagrams in Typst; inspired by but *not* compatible with PlantUML. It is currently in early stages; things *will* still change.

Plum provides the `parse()` and `plum()` functions as entry points, and supports styling through Elembic:



The example above shows a possible model for mathematical expressions. If you're familiar with tools like PlantUML or Mermaid, the mode of creating diagrams will be familiar:

```plum
#[pos(1, 1)]
class Number {
  - value: double {readOnly}
  + evaluate(): double
}

#[pos(2, 0)]
class Binary {
  + evaluate(): double
}

#[pos(2, 1)]
interface BinaryOp {
  + evaluate(a: double, b: double): double
}
…                                            …
Binary o--> (- exprs [2]) Expr
```

One thing Plum is currently lacking is a layout algorithm, so coordinates need to be specified manually. This should change in the future.

The code for rendering the diagram looks like this:

```typc
import plum: elembic as e, diagram, edge, classifier

// let fletcher know about the marks used in Plum UML's edges
plum.add-marks()
// do some general styling
show: e.show_(diagram, it => { set text(0.8em, font: ("FreeSans",)); it })
show: e.set_(edge, stroke: 0.5pt)

// let diagram-src = ...
plum.plum(diagram-src)
```

The central interface `Expression` is highlighted; note that the definition of the diagram and the styling is separated:

```plum
#[pos(1, 0)]
interface Expression as Expr {
  + evaluate(): double
}
```

```typc
show: e.cond-set(
  classifier.with(name: [Expression]),
  fill: color.olive.lighten(50%),
)
```

II Module reference

II.a plum

- `parse()`
- `plum()`

`parse(src: str content) -> dict`

Parses a diagram via a WASM plugin.

```typ
#plum.parse("class Foo")
```

```
(
  classifiers: ((kind: "class", name: "Foo"),),
  edges: (),
)
```

~~~typ
#plum.parse(```
interface Bar
exception Baz
```)
~~~

```
(
  classifiers: (
    (abstract: true, kind: "interface", name: "Bar"),
    (
      kind: "class",
      name: "Baz",
      stereotypes: ("exception",),
    ),
  ),
  edges: (),
)
```

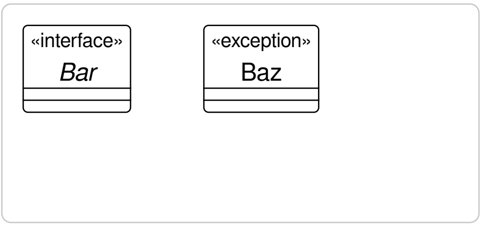
Parameters:

`src` (`str` or `content`) – the expression to parse; may be a `raw` element

`plum(src: str content) -> content`

Parses and processes a diagram.

~~~typ
#plum.plum(```
#[pos(0, 0)]
interface Bar
#[pos(1, 0)]
exception Baz
```)
~~~



The generated diagrams can be styled through the elements described in the following sections using Elembic.

Parameters:

`src` (`str` or `content`) – the expression to parse; may be a `raw` element

II.b `diagram`

- `diagram`

`diagram`

A custom element representing a plum diagram

Fields:

`classifiers` (`array` = `()`) – the classes, interfaces, etc. in the diagram

`edges` (`array` = `()`) – the dependencies, associations, etc. in the diagram

II.c `classifier`

- `divider()`
- `stereotypes`
- `name`

- `member`
- `attribute`
- `operation`

- `classifier`

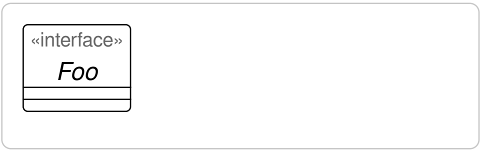
`divider() -> content`

A divider separating sections in a classifier; usually between `attributes` and `operations`.

`stereotypes`

The element that shows stereotypes above a classifiers name.

```typ
#show: e.show_(stereotypes, it => {
  set text(gray.darken(40%)); it
})
#plum.plum("#[pos(0, 0)] interface Foo")
```



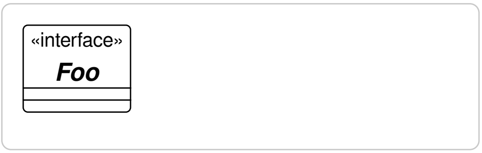
Fields:

`children` (`array`) – the stereotypes of the classifier

name

The element that shows a classifiers name.

```typ
#show: e.show_(name, it => {
  set text(weight: "bold"); it
})
#plum.plum("#[pos(0, 0)] interface Foo")
```



Fields:

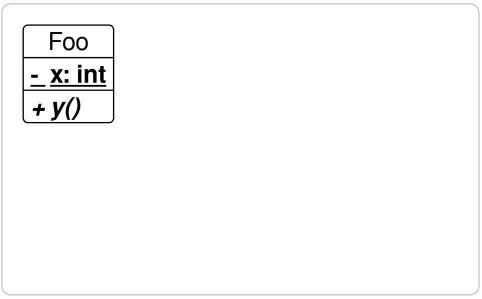
`body` (`content`) – the name of the classifier

member

A member entry of a classifier. Usually, this will contain an `attribute` or `operation`.

The member element shows the visibility modifier and styles the text according to the `static` and `abstract` fields.

```typ
#show: e.show_(member, it => {
  set text(weight: "bold"); it
})
#plum.plum(```
#[pos(0, 0)] class Foo {
  - static x: int
  + abstract y()
}```)
```



Fields:

`body` (`content`) – usually an attribute or operation

`visibility` (`content` = `[]`) – the visibility modifier

`static` (`boolean` = `false`) – the member is underlined if true

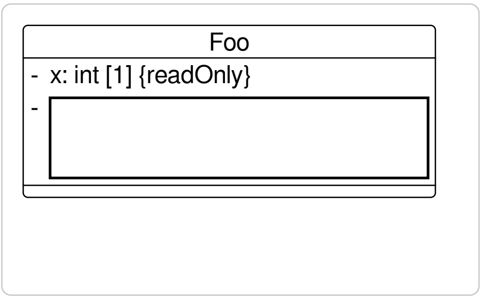
`abstract` (`boolean` = `false`) – the member is italicized if true

`visibility-width` (`length` = `0pt`) – the width of the visibility modifier for alignment

attribute

An attribute. Usually, this will be contained in a `member`.

```
#show: e.show_(attribute.with(name: [y]), it=>{
  rect(width: 5cm) // fill in the blanks
})
#plum.plum(```
#[pos(0, 0)] class Foo {
  - x: int [1] {readOnly}
  - y: int
}```)
```
typ



Fields:

`name` (`content`) – the name of the attribute

`type` (`none or content` = `none`) – the data type of the attribute

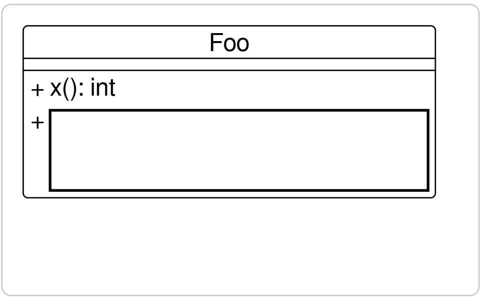
`multiplicity` (`none or content` = `none`) – how many values the attribute contains

`modifiers` (`array` = `()`) – modifiers such as readOnly or invariants

`operation`

An operation. Usually, this will be contained in a `member`.

```
#show: e.show_(operation.with(name: [y]), it=>{
  rect(width: 5cm) // fill in the blanks
})
#plum.plum(```
#[pos(0, 0)] class Foo {
  + x(): int
  + y(): int
}```)
```
typ



Fields:

`name` (`content`) – the name of the operation

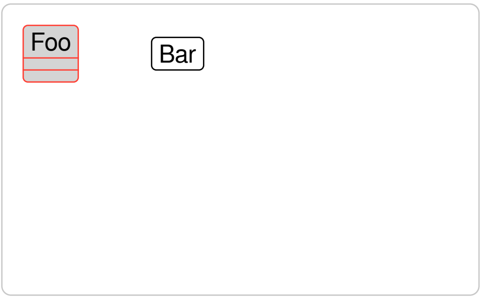
`parameters` (`array` = `()`) – the parameters of the operation; dictionaries consisting of name and optional type

`return-type` (`none or content` = `none`) – the return type of the operation

`classifier`

A class, interface or similar element in an UML class diagram

~~~typ
#show: e.cond-set(classifier.with(name: [Foo]),
  stroke: red, fill: gray.lighten(50%))
#show: e.cond-set(classifier.with(name: [Bar]),
  empty-sections: false)
#plum.plum(```
#[pos(0, 0)] class Foo
#[pos(1, 0)] class Bar
```)
~~~



Fields:

`name` (`content`) – the name of the classifier

`id` (`auto, string or label` = `auto`) – an ID for the classifier, e.g. as a shorthand for a long name

`position` (`any` = `auto`) – the position of the classifier in the diagram; auto can currently not be rendered!

`abstract` (`auto or boolean` = `auto`) – whether the classifier is abstract; interfaces are abstract by default

`final` (`boolean` = `false`) – whether the classifier is final

`stereotypes` (`array` = `()`) – the classifier's stereotypes; interface is added automatically

`kind` (`string` = `"class"`) – the classifier's kind, e.g. class, interface, exception

`members` (`array` = `()`) – the members of the classifier; usually member instances and dividers

`visibility-width` (`length` = `0.8em`) – how much space members should reserve on the left for visibility modifiers

`empty-sections` (`boolean` = `true`) – whether to show or collapse empty sections, i.e. if there are no attributes or operations

`stroke` (`none or stroke` = `0.5pt`) – the stroke for the classifier border and dividers

`fill` (`none, color, gradient or tiling` = `none`) – the fill for the classifier

`radius` (`relative length or dictionary` = `0% + 2pt`) – the border radius for the classifier

II.d edge

- `add-marks()`
- `MARKS`

- `association-end-multiplicity`
- `association-end-role`

- `edge`

`add-marks() -> content`

addd plum-specific marks to Fletcher

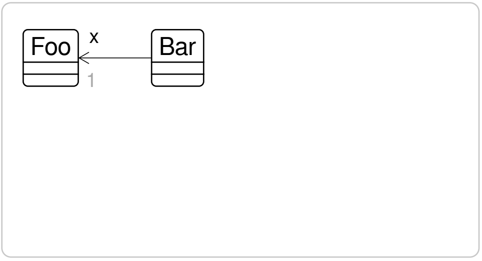
`MARKS: dictionary`

The custom Fletcher marks that Plum defines; can be registered by calling `add-marks()`.

association-end-multiplicity

A multiplicity specifier on one end of an association.

~~~typ
#show: e.show_(association-end-multiplicity,
  it => { set text(gray); it })
#plum.plum(```
#[pos(0, 0)] class Foo
#[pos(1, 0)] class Bar
Foo (x [1]) <-- Bar
```)
~~~



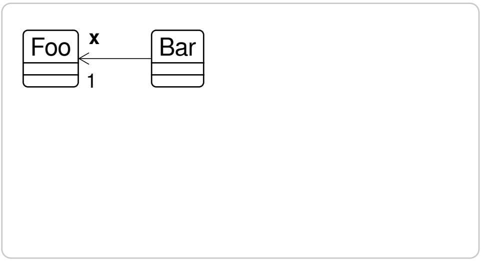
Fields:

`multiplicity` (`content`) – the multiplicity of the association end

association-end-role

A role specifier on one end of an association.

~~~typ
#show: e.show_(association-end-role,
  it => { set text(weight: "bold"); it })
#plum.plum(```
#[pos(0, 0)] class Foo
#[pos(1, 0)] class Bar
Foo (x [1]) <-- Bar
```)
~~~



Fields:

`name` (`content`) – the role name of the association end

`visibility` (`content` = `[]`) – the visibility of the association

`static` (`boolean` = `false`) – whether this is a static association (technically invalid)

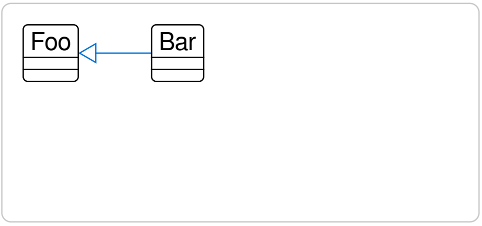
`type` (`none or content` = `none`) – the data type of the association

`modifiers` (`array` = `()`) – modifiers such as readOnly or invariants

edge

An edge between two `classifier`s; can represent associations, dependencies, etc.

~~~typ
#show: e.set_(edge, stroke: blue+0.5pt)
#plum.plum(```
#[pos(0, 0)] class Foo
#[pos(1, 0)] class Bar
Foo <|-- Bar
```)
~~~



Fields:

`a` (`string or label`) – the ID (or name) of the first edge end

`b` (`string or label`) – the ID (or name) of the second edge end

`kind` (`dictionary`) – a dictionary with more information on the edge; at minimum, the type must be defined

`via` (`array` = `()`) – an array of coordinates through which the edge should go (instead of a straight line)

`bend` (`none or float` = `none`) – an angle by which to bend the edge (instead of a straight line)

`stroke` (`none or stroke` = `0.3pt`) – the stroke to use for the edge