

txt2tex Reference

Write math like you're at a whiteboard. Get L^AT_EX you can hand in.

<https://github.com/jmf-pobox/txt2tex>

Document Structure

Single-line directives.

=== Title ===	<code>\section *{Title}</code>
** Solution N **	Bold heading with spacing
(a) Text	Part label
TITLE: My Doc	Document title (<code>\title {}</code>)
SUBTITLE: Sub	Document subtitle
AUTHOR: Name	Author name
DATE: 2026	Date string
INSTITUTION: Univ	Institution line
BIBLIOGRAPHY: f.bib	BibTeX file
BIBLIOGRAPHY_STYLE: plainnat	Citation style
CONTENTS:	Table of contents
PARTS: inline	Parts formatting style
PAGEBREAK:	<code>\newpage</code>
LINEBREAK:	<code>\medskip</code> (gap, no new page)
TEXT: prose here	Smart prose (auto-detects formulas)
PURETEXT: raw & text	Verbatim prose; escapes L ^A T _E X specials
LATEX: <code>\cmd</code>	Raw L ^A T _E X passthrough (single-line)
[cite key]	<code>\citep {key}</code> (in TEXT blocks)

Identifiers

Decorations.

<code>count'</code>	<i>count'</i> (after-state)
<code>in?</code>	<i>in?</i> (input)
<code>out!</code>	<i>out!</i> (output)
<code>x?'</code>	<i>x?'</i> (combinations allowed)
<code>'sunk'</code>	'sunk' (string literal)

Reserved words.

None of the keywords below can be used as an identifier name or carry decorations. Watch out for the short ones (id, dom, P, F, as).

<code>land, lor, lnot</code>	propositional
<code>forall, exists, exists1, mu, lambda</code>	quantifiers / binders
<code>elem, notin, subset, subseteq, psubset</code>	set membership / subset
<code>union, intersect, cross, bigcup, bigcap</code>	set operations
<code>dom, ran, inv, id, comp</code>	relation operators
<code>P, P1, F, F1</code>	power / finite set
<code>mod</code>	arithmetic
<code>if, then, else, otherwise</code>	conditional
<code>schema, axdef, gendef, zed, syntax</code>	block-paragraph headers
<code>given, where, end</code>	paragraph delimiters
<code>Delta, Xi, theta, defs, shows</code>	schema operators
<code>hide, project</code>	schema calculus
<code>pk</code>	txt2tex extension (primary-key annotation)
<code>sigma, pi, join, div, group, ungroup</code>	relational algebra
<code>Count, Sum, Avg, Min, Max, Median, as</code>	aggregators
<code>filter, bag_union, bag_diff</code>	sequence / bag
<code>ARGUE, EQUIV, EQUAL, INFRULE, PROOF</code>	proof-paragraph markers
<code>TEXT, PURETEXT, LATEX, TRUTH, TABLE</code>	prose / table markers

Numbers & Arithmetic

N	N (natural numbers)
Z	Z (integers)
N ₁	N ₁ (positive naturals)
n + m / n - m	addition / subtraction
n * m / n div m	multiplication / integer division
n mod m	modulo
n < m / n > m	strict ordering
n <= m / n >= m	non-strict ordering
n = m / n /= m	equality / inequality

Propositional Logic

p land q	$p \wedge q$
p lor q	$p \vee q$
lnot p	$\neg p$
p => q	$p \Rightarrow q$
p <=> q	$p \Leftrightarrow q$

Note: Use **land**, **lor**, **lnot**. English **and/or/not** are not supported.

Example — truth table (validate De Morgan).

You write:

TRUTH TABLE:							
p	q	p land q	lnot (p land q)	lnot p	lnot q	lnot p lor lnot q	
T	T	T	F	F	F	T	
T	F	F	T	F	T	T	
F	T	F	T	T	F	T	
F	F	F	T	T	T	T	

You get:

p	q	$p \wedge q$	$\neg (p \wedge q)$	$\neg p$	$\neg q$	$\neg p \vee \neg q$
T	T	T	F	F	F	F
T	F	F	T	F	T	T
F	T	F	T	T	F	T
F	F	F	T	T	T	T

Build intermediate columns step by step. Columns 4 and 7 match on every row, so $\neg (p \wedge q) \Leftrightarrow \neg p \vee \neg q$ (De Morgan).

Exponentiation. The \wedge operator is fuzz *iter*: it composes a relation with itself. It does NOT square a number. For powers of numbers, multiply directly.
You write:

R^3	(relation iterated three times)
$x * x$	(square of a number)

You get:

$R^3 \quad x * x$

Example — EQUIV chain.

You write:

EQUIV:	
lnot (p land (q lor r))	
lnot p lor lnot (q lor r)	[De Morgan]
lnot p lor (lnot q land lnot r)	[De Morgan]
(lnot p lor lnot q) land	
(lnot p lor lnot r)	[distribute]

You get:

$\neg (p \wedge (q \vee r))$	
$\Leftrightarrow \neg p \vee \neg (q \vee r)$	[De Morgan]
$\Leftrightarrow \neg p \vee (\neg q \wedge \neg r)$	[De Morgan]
$\Leftrightarrow (\neg p \vee \neg q) \wedge (\neg p \vee \neg r)$	[distribute]

EQUIV: and its alias **ARGUE**: join steps with \Leftrightarrow . Use for predicate equivalences.

Example — EQUAL chain.

You write:

EQUAL:	
$\#(s \wedge \langle a, b, c \rangle)$	
$\#s + \# \langle a, b, c \rangle$	[#-concat]
$\#s + 3$	[literal length]
$3 + \#s$	[+ commutative]

You get:

$\#(s \wedge \langle a, b, c \rangle)$	
$= \#s + \# \langle a, b, c \rangle$	[#-concat]
$= \#s + 3$	[literal length]
$= 3 + \#s$	[+ commutative]

EQUAL: joins steps with $=$. Use when steps are arithmetic/set-valued, not propositional equivalences.

Sets & Types

x elem S	$x \in S$
x notin S	$x \notin S$
A subset B	$A \subseteq B$
A psubset B	$A \subset B$ (proper subset)
A union B	$A \cup B$
A intersect B	$A \cap B$
A setminus B	$A \setminus B$
bigcup S	$\bigcup S$
bigcap S	$\bigcap S$
power S	$\mathbb{P} S$
P1 S	$\mathbb{P}_1 S$ (non-empty)
F S	$\mathbb{F} S$
F1 S	$\mathbb{F}_1 S$ (non-empty finite)
A cross B	$A \times B$
{x : S P}	Set comprehension
n..m	$n..m$ (integer range)
# S	$\#S$ (cardinality / length)

Nested prefix operators. Prefix-generic operators (`seq`, `P`, `dom`, `ran`, `bigcup`, ...) take a single atomic operand. The engine wraps a nested prefix call in parentheses automatically: `seq (P X) → \seq~(\power X)`.

Set comprehension:

You write:

{x : N | x > 0 land x < 5}

You get:

$$\{x : \mathbb{N} \mid x > 0 \wedge x < 5\}$$

Predicate Logic

forall x : S P	$\forall x : S \bullet P$
exists x : S P	$\exists x : S \bullet P$
exists_1 x : S P	$\exists_1 x : S \bullet P$ (unique)
lambda x : S E	$\lambda x : S \bullet E$
mu x : S P	$\mu x : S \bullet P$ (definite desc.)
if p then e1 else e2	conditional expression

Bullet form: $x : S \mid P$ means $x : S \bullet P$ (bullet is implied). The vertical bar is the quantifier separator.

Tuple-pattern:

You write:

forall (x, y) : N cross N |
x + y >= 0

You get:

$$\forall (x, y) : \mathbb{N} \times \mathbb{N} \bullet x + y \geq 0$$

Multi-decl: separate variable groups with ; inside a quantifier: `forall x : A; y : B | P`.

Schema-text scope: a bare [d : T | P] in a quantifier or function acts as an inline schema text.

Relations

A <-> B	$A \leftrightarrow B$ (relation type)
x -> y	$(x \mapsto y)$ (maplet)
dom R / ran R	$\text{dom } R / \text{ran } R$
id	identity relation
inv R / R~	R^{-1} (inverse; postfix or prefix)
A dres R / A < R	$A \triangleleft R$ (domain restrict)
A dsub R / A << R	$A \triangleleft\!\!\!\triangleleft R$ (domain subtract)
R rres B / R > B	$R \triangleright B$ (range restrict)
R rsub B / R » B	$R \triangleright\!\!\!\triangleright B$ (range subtract)
R oplus S / R ++ S	$R \oplus S$ (relational override)
R o9 S	$R \circ_9 S$ (forward comp.)
R comp S	$R \circ_8 S$ (backward comp.)
R(A)	$R[A]$ (relational image)
shows	\vdash (turnstile / judgment)

o9 vs comp: `o9 → \semi` (forward, $(R; S)(x) = S(R(x))$). `comp → \comp` (backward, $(R \circ S)(x) = R(S(x))$).

Example — domain restrict.

You write:

```
{1, 2} dres {1 |-> a, 2 |-> b, 3 |-> c}
```

You get:

$$\{1, 2\} \triangleleft \{1 \mapsto a, 2 \mapsto b, 3 \mapsto c\}$$

Functions

A +-> B	$A \mapsto B$ (partial)
A -> B	$A \longrightarrow B$ (total)
A >+> B / A - > B	$A \mapsto B$ (partial injection)
A >-> B	$A \mapsto B$ (total injection)
A +-» B	$A \twoheadrightarrow B$ (partial surjection)
A -» B	$A \twoheadrightarrow B$ (total surjection)
A >-» B	$A \twoheadrightarrow B$ (bijection)
A 77-> B	$A \mapsto B$ (finite partial)
f(x)	function application
lambda x : T f(x)	$\lambda x : T \bullet f(x)$
f ++ g	$f \oplus g$ (override)
f~	f^{-1} (postfix inverse)

Example — lambda.

You write:

```
double == lambda n : N | n * 2
```

You get:

$$double == \lambda n : \mathbb{N} \bullet n * 2$$

Sequences & Bags

<> / <a, b>	$\langle \rangle$ / $\langle a, b \rangle$
s ^ t	$s \frown t$ (concat; space before ^ required)
seq S / seq1 S	$\text{seq } S$ / $\text{seq}_1 S$
iseq S	$\text{iseq } S$ (injective sequences)
# s	$\#s$ (length)
head s / tail s	first / rest
last s / front s	last / all-but-last
rev s	reverse
s(i)	index (1-based)
s filter A	$s \upharpoonright A$ (filter)
bag S	$\text{bag } S$
[[a, b]]	$\llbracket a, b \rrbracket$ (bag literal)
b1 bag_union b2	$b_1 \uplus b_2$
b1 bag_diff b2	$b_1 \ominus b_2$

Concat: write <a> ^ (space before ^). No space → relation iteration (\bsup). **bag_diff:** emits \uminus — bag subtraction clamped at zero.

Z

A Z specification is a sequence of paragraphs. Each paragraph either introduces names or constrains them. The kinds below cover the building blocks; **schema** (the most-used kind) gets its own section.

Given (carrier) type. An opaque set whose elements are not enumerated — used when identity matters but structure does not.

given A, B $[A, B]$

Free type. A named type with a fixed enumeration of constructors. Branches can carry data via $\langle \dots \rangle$.

Status ::= active | inactive $Status ::= active \mid inactive$

Abbreviation. Binds a name to an expression for reuse — no new type.

MaxSize == 100 $MaxSize == 100$

Axiomatic definition. Introduces global constants constrained by predicates. Form: **axdef** declarations **where** predicates **end**. (See *Generic schemas* for the **gendif** [X] variant.)

You write:

```
axdef
  MaxSize : N
  Reset : N -> N
where
  MaxSize > 0
  Reset(MaxSize) = 0
end
```

You get:

$MaxSize : \mathbb{N}$
$Reset : \mathbb{N} \longrightarrow \mathbb{N}$
$MaxSize > 0$
$Reset(MaxSize) = 0$

Source-level containers. The forms above stand alone (the engine wraps them in **zed**). Use explicit containers to bundle several paragraphs or align multiple free types.

zed ...end bundle several standalone paragraphs (unboxed)
syntax ...end column-aligned multi-type free definitions

Example — zed bundle. Combines a *given type*, a *free type*, and an *abbreviation* in one unboxed block.

You write:

```
zed
  given Entry
  Status ::= active | inactive
  MaxSize == 100
end
```

You get:

$[Entry]$

$Status ::= active \mid inactive$

$MaxSize == 100$

Example — syntax (aligned free types). Two mutually-recursive free types defined together with their ::= columns aligned.

You write:

```
syntax
  OP ::= plus | minus | times
  Expr ::= const<N> | binop<OP cross Expr>
end
```

You get:

$OP ::= plus \mid minus \mid times$

$Expr ::= const\langle \mathbb{N} \rangle \mid binop\langle OP \times Expr \rangle$

Blank lines between definitions inside **syntax/zed** emit \also (visual group separator).

Schemas

A named schema introduces a signature (declarations) and an invariant (where clause).

You write:

```
schema Counter
  size : N
  nums : seq N
where
  size = # nums
end
```

You get:

$Counter$
$size : \mathbb{N}$
$nums : \text{seq } \mathbb{N}$
$size = \#nums$

Where-clause patterns.

1. Set comprehension.

You write:

```
schema Inventory
  items : seq N
  stock : N
where
  stock = # {x : ran items | x > 0}
end
```

You get:

$Inventory$
$items : \text{seq } \mathbb{N}$
$stock : \mathbb{N}$
$stock = \#\{x : \text{ran } items \mid x > 0\}$

2. Quantifier.

You write:

```
schema NonNeg
  s : seq Z
where
  forall i : 1..#s | s(i) >= 0
end
```

You get:

$NonNeg$
$s : \text{seq } \mathbb{Z}$
$\forall i : 1.. \#s \bullet s(i) \geq 0$

Generic schemas.

You write:

```
gendef [X]
  empty : F X
where
  empty = {}
end
```

You get:

$[X]$
$empty : F X$
$empty = \{\}$

[X] is the type parameter; instantiate at use with `empty[T]`.

Schema Operations

Inclusion operators.

SName	bare inclusion
Delta Counter	$\Delta Counter$ (before/after-state pair)
Xi Card	$\Xi Card$ (read-only state)
theta S	θS (binding of current state)
theta S'	$\theta S'$ (binding of after-state)

Horizontal definition.

Name	defs	RHS:
Name	defs	Schema $Name \hat{=}$ Schema
N	defs	Delta C $N \hat{=}$ ΔC
N	defs	[d : T P] inline schema text
N[X]	defs	G[X] generic form

Schema calculus operators (RHS of defs).

S[old/new]	rename component
S[a/b, c/d]	multiple renames
S ; T	$S \circ T$ (composition)
S » T	$S \gg T$ (piping)
S hide (x, y)	$S \setminus (x, y)$
S project T	$S \upharpoonright T$

Example — operation schema with Delta.

You write:

```
schema Bump
  Delta Counter
where
  size' = size + 1
  nums' = nums
end
```

You get:

$Bump$
$\Delta Counter$
$size' = size + 1$ $nums' = nums$

Example — schema calculus (composition).

You write:

```
TwoBumps defs Bump ; Bump
```

You get:

$$TwoBumps \hat{=} Bump \circ Bump$$

Example — promotion schema (theta lifts local to global).

Bank has counters : ID \mapsto Counter (partial — only some IDs are bound); promotion lifts a local **Bump** into one entry:

You write:

```
schema PromoteBump
  Delta Bank
  Delta Counter
  key? : ID
where
  key? elem dom counters
  counters(key?) = theta Counter
  counters' = counters ++ {key?  $\mapsto$  theta Counter'}
end
```

You get:

$PromoteBump$
$\Delta Bank$
$\Delta Counter$
$key? : ID$
$key? \in \text{dom } counters$ $counters \text{ key?} = \theta Counter$ $counters' = counters \oplus \{key? \mapsto \theta Counter'\}$

Z Bindings

theta S	θS (binding of S in scope; fuzz-typed)
theta S'	$\theta S'$ (primed-component binding)
{ name == e }	$\langle name == e \rangle$ (literal — fuzz-incompatible)
{ a == e, b == f }	multiple components
{ }	empty binding

Multi-typed comprehension with binding:

You write:

```
{ t : Track; a : Album |
  t.albumId = a.albumId .
  { | trackId == t.trackId | } }
```

You get:

$$\{ t : Track; a : Album \mid t.albumId = a.albumId \\ \bullet \langle trackId == t.trackId \rangle \}$$

Use **;** to separate variable-type pairs inside { }. Binding components are **comma**-separated.

Fuzz note: the { |...| } literal binding is fuzz-incompatible — the engine emits it outside any Z paragraph so it renders without participating in fuzz type-checking. Use **theta** when you need a fuzz-typed binding inside a schema.

Proofs

INFRULE: — named inference rule display:

You write:

```
INFRULE:
premise_1
premise_2
---
conclusion [Rule name]
```

You get:

$$\frac{\text{premise}_1 \quad \text{premise}_2}{\text{conclusion}} \text{ Rule name}$$

INFRULE: renders a labelled rule schema. Use for displaying rule definitions (not proofs). The three-dash line `--` separates premises from conclusion.

How proof trees work (PROOF:):

The source is **conclusion-first** (rendered tree reads bottom-up). Premises are indented under their conclusion. A rule with two or more premises requires `::` on each premise — without it, indented lines collapse into a *linear chain* (each line becomes the sole premise of the line above) and the extra premises are silently dropped. Unary rules take a single indented child with no `::`.

Proof tree syntax:

indent	Premises indented under conclusion
[rule]	Justification label
[rule from <i>n</i>]	Discharge assumption <i>n</i>
[<i>n</i>] expr [assumption]	Boxed assumption labelled <i>n</i>
expr [from <i>n</i>]	Leaf reference to assumption <i>n</i>
::	Premise of a multi-premise rule
case <i>X</i> :	Branch in a lor elim case analysis

Rules: `land intro`, `land elim 1/2`, `lor intro 1/2`, `lor elim`, `=> intro/elim`, `<=> intro/elim`, `lnot intro/elim`, `forall intro/elim`, `exists intro/elim`.

Modus ponens (binary — requires `::` on each premise):

You write:

```
PROOF:
q [=> elim]
:: p => q
:: p
```

You get:

$$\frac{p \Rightarrow q \quad p}{q} \Rightarrow \text{-elim}$$

\wedge -intro (binary):

You write:

```
PROOF:
p land q [land intro]
:: p
:: q
```

You get:

$$\frac{p \quad q}{p \wedge q} \wedge \text{-intro}$$

\Rightarrow -intro with discharge (use from N):

You write:

```
PROOF:
p => p [=> intro from 1]
[1] p [assumption]
:: p [from 1]
```

You get:

$$\frac{\lceil p \rceil^{[1]}}{p \Rightarrow p} \Rightarrow \text{-intro}^{[1]}$$

`[1] p [assumption]` declares the discharge binder. `p [from 1]` marks every leaf use. `[=> intro from 1]` discharges it.

\vee -elim / case analysis (ternary — disjunction + two cases):

You write:

```
PROOF:
(p lor q) => r [=> intro from 1]
[1] p lor q [assumption]
:: r [lor elim from 1]
:: p lor q [from 1]
case p:
:: r [...derive r from p...]
case q:
:: r [...derive r from q...]
```

You get:

$$\frac{p \vee q \quad \frac{\lceil p \rceil^{[1]}}{r} \quad \frac{\lceil q \rceil^{[1]}}{r}}{r} \vee \text{-elim}$$

lor elim from N discharges the disjunction's assumption `[N]`; inside each **case X:** block, the case formula is referenced as **X [from N]**.

Relational Database Notation

Primary key annotation.

Mark a primary-key attribute with **pk** in a named schema body. The annotation sits outside the Z box so fuzz type-checks cleanly.

You write:

```
schema Book
  pk bookId : BookId
  isbn      : ISBN
  pages     : N
end
```

You get:

Book

bookId : *BookId*
isbn : *ISBN*
pages : \mathbb{N}

$$\text{PK}(\text{Book}) = \{bookId\}$$

Composite: two **pk** lines $\Rightarrow \text{PK}(S) = \{a, b\}$. Comma-separated names on one **pk** line also work. **pk** is rejected in **axdef/gendef**/inline schema text.

Relational algebra.

<code>sigma[p] (R)</code>	$\text{Restrict}_p(R)$ — restrict
<code>pi [A,B] (R)</code>	$\text{Project}_{A,B}(R)$ — project
<code>R[B/A]</code>	$R[B/A]$ — rename
<code>R join S</code>	$\text{Join}(R, S)$ — natural join
<code>R join [p] S</code>	$\text{Join}_p(R, S)$ — theta-join
<code>R div S</code>	$R \text{ div } S$ — division

Fuzz note: relational-algebra operators are fuzz-incompatible — the engine emits them outside any Z paragraph so they render without participating in fuzz type-checking. Do not wrap algebra inside **zed/axdef/schema** blocks.

FK predicate form.

Foreign-key constraints are expressed as **axdef** predicates:

You write:

```
axdef
where
  (R, S) |-> {a |-> b} elem FK
end
```

You get:

$(R, S) \mapsto \{a \mapsto b\} \in FK$

GROUP & UNGROUP.

Date’s nested-relation operators.

<code>R group ({A,B} as x)</code>	$R \text{ GROUP}(\{A, B\} \text{ AS } x)$
<code>R ungroup x</code>	$R \text{ UNGROUP } x$

You write:

```
Members group ({name,email} as contact)
Members ungroup contact
```

You get:

Members GROUP (*{name, email}* AS *contact*)

Members UNGROUP *contact*

Fuzz note: GROUP and UNGROUP are fuzz-incompatible — the engine emits them outside any Z paragraph so they render without participating in fuzz type-checking. Do not wrap GROUP/UNGROUP inside **zed/axdef/schema** blocks.

GROUP aggregator form.

The aggregator keywords are **Count**, **Sum**, **Avg**, **Min**, **Max**, **Median**. Each takes one attribute name and an alias. The regroup form (**{...}** as alias) and the aggregator form are mutually exclusive in a single **group** expression.

<code>R group (Count(x) as n)</code>	$R \text{ Group}(\text{Count}(x) \text{ as } n)$
<code>R group (Sum(x) as n)</code>	$R \text{ Group}(\text{Sum}(x) \text{ as } n)$
<code>R group (Avg(x) as n)</code>	$R \text{ Group}(\text{Avg}(x) \text{ as } n)$
<code>R group (Min(x) as n)</code>	$R \text{ Group}(\text{Min}(x) \text{ as } n)$
<code>R group (Max(x) as n)</code>	$R \text{ Group}(\text{Max}(x) \text{ as } n)$
<code>R group (Median(x) as n)</code>	$R \text{ Group}(\text{Median}(x) \text{ as } n)$

You write:

```
Sales group (Count(orderId) as orderCount,
             Sum(amount) as totalRevenue)
```

You get:

Sales Group(*Count(orderId)* as *orderCount*,
Sum(amount) as *totalRevenue*)

Multiple aggregators are comma-separated. Each accepts exactly one attribute identifier — no nested aggregators, no compound expressions.

Operator Precedence

Tightest binding at the top. Same level groups are left-associative unless noted. Use parentheses to override.

- 1 postfix `~`, `+`, `*` (inverse, transitive closures); `R^n` iter (no space around `^`)
- 2 prefix `dom`, `ran`, `#`, `P`, `F`, `seq`, `bigcup`, `bigcap`, `inv`, `id`
- 3 atom-level postfix: function application `f(x)`, relational rename `R[B/A]`, relational atoms `sigma[...]`, `pi[...]`
- 4 `*`, `mod` (multiplicative)
- 5 `+`, `-` (binary); `s ^ t` concat (spaces required); `filter`, `bag_union`, `bag_diff`
- 6 `range ..`
- 7 `intersect`, `setminus` (`\`)
- 8 `cross`, `join`, `div`, `group`, `ungroup` (cross-product / relational)
- 9 `union`, `++` (override)
- 10 relational infix `|->`, `<|`, `|>`, `<|`, `|>`, `o9`, `comp`
- 11 function / relation arrows `->`, `<->`, `+>`, `>->`, `>+>`, `-|>`, `>->`, `->`, `+>`, `77->`
- 12 `elem`, `notin`, `subset`, `subseteq`, `psubset`
- 13 comparison `<`, `>`, `<=`, `>=`, `=`, `!=`
- 14 `lnot`
- 15 `land`
- 16 `lor`
- 17 `=>` (right-associative)
- 18 `<=>` (left-associative; logically commutative)
- 19 quantifiers `forall`, `exists`, `exists1`, `mu`, `lambda` (loosest)

Schema-calculus operators appear only on the RHS of `defs`; they bind looser than expression-level operators and aren't part of this table. Internal order, tightest to loosest: `hide` / `project` (postfix) `<` ; composition `< >>` piping.