AtoState Developer's Guide

Contents

Parsing 1
Testing 2
  User-Facing API 2
  Internal API 2
Errors 2
API 3
  AtoState (User-Facing) 3
    atostate 3
    atoterm 3
  Parser 4
    AtoState-Specific Parsers 4
    Generic Parsers 4
    Abstract Syntax Tree 11

Parsing

Parsing is based on parser combinators.

For our purposes a parser is a function which takes a single argument—the input as a string—consumes zero or more characters from the **start** of the string, and returns the remainder of the string alongside a representation of the parser data as an array with two elements. If the parser does not find a match, it should return the original input and `none` as an array with two elements. For example, a simple parser which consumes any single character from the input would be defined as[1]:

```
#let any-char(input) = if input.len() > 0 {
  (input.slice(1), input.at(0))
} else {
  (input, none)
}
```

```
#any-char("abc")
```

```
("bc", "a")
```

```
#any-char("")
```

```
("", none)
```

A combinator, on the other hand, is a function which takes any arguments (although usually parsers) and *produces a single parser*. A pair combinator, which applies two parsers one after the other returning their results as an array, would be defined as:

```
#let pair(first, second) = input => {
  let bookmark = input
  let (input, first-output) = first(input)
  let (input, second-output) = second(input)
  if first-output == none or second-output == none {
    (bookmark, none)
  } else {
    (input, (first-output, second-output))
  }
}
```

[1] Because our grammar only allows ASCII characters, I will assume all inputs are ASCII.

```
#pair(any-char, any-char)("abc")
#pair(any-char, any-char)("a")
#pair(any-char, any-char)("")
```

```
("c", ("a", "b"))
("a", none)
("", none)
```

Note how we've used bookmarking here. If `second` had failed and we just returned `input` as it had been returned from `first`, then the parser would advance even though it had failed. This would violate our definition of a parser!

Note also that we mark a failure to parse using `none`, mostly due to the fact that typst doesn't have any error recovery mechanisms at this point. This could, in principle, lead to ambiguities if a parser wanted to return `none` on success. For a fully generic parser combinator library this would be a big issue, but for our purposes just don't return `none` if the parser was successful.

Parser combinators allow one to compose parsers in a way that *can be* very readable, but don't go crazy! Things can get real slow when you use lots of composition and the trade-offs we've had to make can cause readability to become a big concern, but the good thing about them is that you can build new parsers or combinators easily without having to use composition[2]. If you find yourself nesting combinators more than two levels deep, you probably want to just want to write a function to apply them yourself.

Testing

User-Facing API

Testing of user-facing functions should be comprehensive. Both happy paths—inputs that you expect to succeed—and failure modes—inputs that you think should fail—should have enough tests that you reasonably think all code paths have been covered.

Part of what we're testing when we test user-facing functions is that we're not accidentally breaking the API. Any *intentional* change to the API, on the other hand, should be considered a big deal, and the effort that would be required to change all the tests should reflect that.

Internal API

The happy paths of internal APIs are much less important, they are internal partly *so that* we don't have to worry so much about maintaining a consistent API. There should be some testing of the happy paths, as an early indicator if the function breaks, but the user-facing API should be tested enough that any breakages become apparent quickly.

Failure modes should still be tested comprehensively, however. This includes both handled failures—where the function itself can handle the error and keep on going—and unhandled failures—where the error is signalled to the calling function[3]

Errors

Use structure to describe syntax errors where possible, instead of

```
error(input, "fraction of positive numbers")
```

use:

```
error(input, "positive number '/' positive number")
```

[2] The `delim` combinator, for example, could be made with the `all` and `map` combinators. But we avoid that for readability and performance purposes.

[3] For now, this is almost always through a `panic` which will necessarily reach the user, or returning `none`.

API

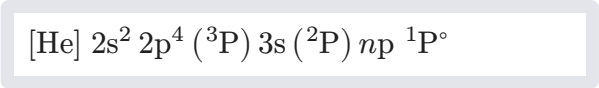
AtoState (User-Facing)

- atostate()
- atoterm()

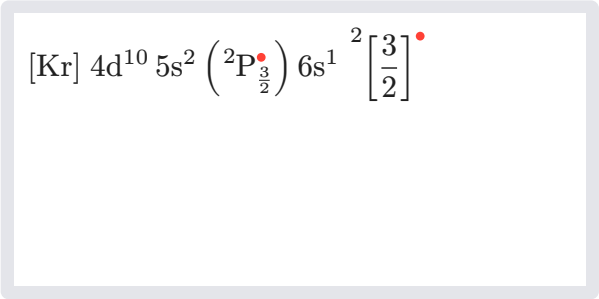
atostate

Parse and render an atomic state.

```
$ #atostate("[He] 2s2.2p4.(3P).3s.(2P).np:1P*") $
```



```
$
  #atostate(
    "[Kr] 4d10.5s2.(2P<3/2>*).6s:2[3/2]*",
    parity-marker: circle(
      radius: 0.2em,
      fill: red,
    ),
    display-single-occupation: true,
  )
$
```



Parameters

```
atostate(
  state: str,
  parity-marker: str content,
  display-single-occupation: bool
) -> content
```

state `str`

The atomic state to render.

parity-marker `str` or `content`

The symbol to use to indicate that a term has odd parity.

Default: `sym.circle.tiny`

display-single-occupation `bool`

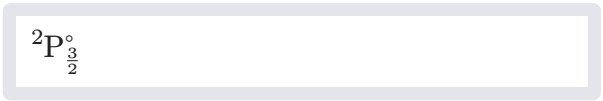
Whether to display the occupation number of an orbital that only has one electron.

Default: `false`

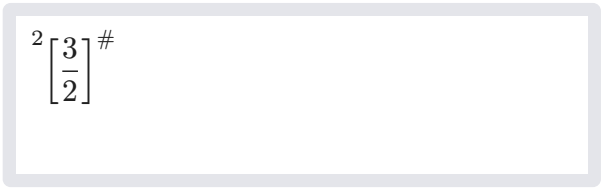
atoterm

Parse and render a single term symbol.

```
$ #atoterm("2P<3/2>*") $
```



```
$
  #atoterm(
    "2[3/2]*",
    parity-marker: "#",
  )
$
```



Parameters

```
atoterm(
  term: str,
  parity-marker: str content
) -> content
```

term `str`

The term symbol to render.

parity-marker `str` or `content`

The symbol to use to indicate that the term has odd parity.

Default: `sym.circle.tiny`

Parser

AtoState-Specific Parsers

Generic Parsers

- all()
- alt()
- delim()
- ensure()
- ensured-tag()
- error()
- map()
- map-or()
- rgx()
- sep1()
- seq()
- seq-comp()
- tag()

Variables

- whitespace
- integer
- negative-integer

all

Apply a list of parsers one after the other, *if all succeed*.

If any parser returns none then this parser will stop and return none.

```
#all(tag("ab"), tag("12"), tag("yz"))("abyz")

#all(tag("ab"), tag("12"), tag("yz"))("ab12yz")

#all(tag("ab"), tag("12"), tag("yz"))("ab12")
```

```
("abyz", none)
("", ("ab", "12", "yz"))
("ab12", none)
```

Parameters

```
all(..parsers: func) -> func
```

..parsers `func`

The parsing functions to apply.

alt

Try a list of parsers until one succeeds.

There is no ambiguity checking, make sure you order the parsers in such a way that a later parser can't be matched by an earlier parser.

```
#let parser = alt(tag("123"), tag("abc"))

#parser("123 abc")

#parser("abc 123")

#parser("xyz 123")
```

```
(" abc", "123")
(" 123", "abc")
("xyz 123", none)
```

Parameters

```
alt(..parsers: func) -> func
```

..parsers `func`

The parsing functions to try.

delim

Match a delimited parser.

Applies the first parser and discards it, then applies the second and keeps the result, and finally applies the third and discards it.

Whitespace directly after the left delimiter, or before the right delimiter, is ignored.

```
#delim(tag("("), integer, tag(")"))("( 123 )")
```

```
("", "123")
```

```
#delim(tag("|"), tag("psi"), tag(">"))("| psi >")
```

```
("", "psi")
```

Parameters

```
delim(
  left: func,
  delimited: func,
  right: func
) -> func
```

left `func`

The parser to match the left delimiter.

delimited `func`

The parser to match the delimited contents.

right `func`

The parser to match the right delimiter.

ensure

Ensure that the parser has found its target.

Since parsers are optional by default, this is an easy way to make one mandatory.

```
// Succeeds!
#ensure(tag("1"), "'1'")("123")
// Would panic.
// #ensure(tag("1"), "'1'")("23")
```

```
("23", "1")
```

Parameters

```
ensure(
  parser: func,
  ..expected: (str,)
) -> func
```

parser `func`

The parsing function to wrap.

..expected `(str,)`

Description of what was expected to be at this location.

ensured-tag

Apply the `tag()` parser, and raise a sensible error if it is none.

```
// Succeeds!
#ensured-tag("hello")("hello, world")
// Would fail.
// #ensured-tag("hello")(", world")
```

```
(", world", "hello")
```

Parameters

```
ensured-tag(expected: str) -> func
```

expected `str`

The string that should be matched.

error

Fail with a helpful error message.

Parameters

```
error(
  input: str,
  ..expected: array
)
```

input `str`

Input that caused the error.

..expected `array`

Description of what was expected to be at this location.

Where needed this should be structural, rather than descriptive, e.g.

`error(input, "positive integer '/' positive integer")`

as opposed to:

`error(input, "fraction of positive integers")`

map

Apply a function to the successful result of a parser.

```
#map(tag("123"), int)("123 abc")

#map(tag("123"), int)("abc 123")
```

```
(" abc", 123)

("abc 123", none)
```

Parameters

```
map(
  parser: func,
  transformer: func
) -> func
```

parser `func`

The parsing function to wrap.

transformer `func`

The function to apply to the result.

map-or

Apply a function to the successful result of a parser, or return the given value on the unsuccessful result.

```
#map-or(tag("123"), int, 321)("123 abc")

#map-or(tag("123"), int, 321)("abc 123")
```

```
(" abc", 123)
("abc 123", 321)
```

Parameters

```
map-or(
  parser: func,
  transformer: func,
  other: any
) ->
```

parser `func`

The parsing function to wrap.

transformer `func`

The function to apply to the result.

other `any`

The value to return if the parser fails.

rgx

Match a regular expression and return its match.

Your regex should start with the `^` or `\A` matchers, otherwise unexpected behaviour may occur. This is checked at parse-time, but not perfectly.

```
#rgx(regex("^\d+"))("123 abc")
// This would panic.
// #rgx(regex("\d+"))("abc 123")
```

```
(" abc", "123")
```

Parameters

```
rgx(expression: regex) -> func
```

expression `regex`

The regular expression to match.

sep1

Match at least one element of a separated list.

Optionally use a different parser for the first element.

```
#sep1(
  map(alt(integer, negative-integer), int),
 tag(","),
)("123, -456, 789")
```

```
("", (123, -456, 789))
```

```
#sep1(
  map(alt(integer, negative-integer), int),
 tag(","),
 first: map(integer, int)
)("-123, 456, -789")
```

```
("-123, 456, -789", none)
```

Parameters

```
sep1(
  element: func,
  separator: func,
  first: func
) -> func
```

element `func`

The parser to match elements of the list.

separator `func`

The parser to match the separator.

first `func`

The parser to match the first element of the list.

Default: none

seq

Apply a list of parsers one after the other.

The output list will always be the same length as there are parsers. If a parser fails, `none` will be entered into the list.

```
#seq(tag("ab"), tag("12"), tag("yz"))("abyz")

#seq(tag("ab"), tag("12"), tag("yz"))("ab12yz")

#seq(tag("ab"), tag("12"), tag("yz"))("ab12")
```

```
("", ("ab", none, "yz"))
("", ("ab", "12", "yz"))
("", ("ab", "12", none))
```

Take care! This parser always succeeds, even if all the parsers fail (in which case `none` will be returned for all parsers).

Parameters

```
seq(..parsers: func) -> func
```

..parsers `func`

The parsing functions to apply.

seq-comp

Apply a list of parsers one after the other, returning only the outputs of parsers that succeed.

"comp" is short for "compress".

```
#seq-comp(tag("ab"), tag("12"), tag("yz"))("abyz")

#seq-comp(tag("ab"), tag("12"), tag("yz"))
("ab12yz")

#seq-comp(tag("ab"), tag("12"), tag("yz"))("ab12")
```

```
("", ("ab", "yz"))
("", ("ab", "12", "yz"))
("", ("ab", "12"))
```

Take care! This parser always succeeds, even if all the parsers fail (in which case an empty array will be returned).

Parameters

```
seq-comp(..parsers: func) -> func
```

..parsers `func`

The parsing functions to apply.

tag

Match and return a given string.

The tag must not be an empty string.

```
#tag("hello")("hello, world")
```

```
```

```
(", world", "hello")
```

Parameters

```
tag(expected: str) -> func
```

expected `str`

The string that should be matched.

whitespace

Match a run of whitespace.

```
#whitespace("\n\t a \t\n")

#all(tag("a"), whitespace, tag("b"))("a b")
```

```
("a \t\n", "\n\t ")
("", ("a", " ", "b"))
```

integer

Match a positive integer.

```
#map(integer, int)("123abc")
```

```
("abc", 123)
```

```
#map(
  all(tag("-"), integer),
  o => int(o.join()),
)("-23")
```

```
("", -23)
```

negative-integer

Match a negative integer.

```
#map(negative-integer, int)("-123for")
```

```
("for", -123)
```

Abstract Syntax Tree

- element()
- fraction()
- letter()
- number()
- orbital()
- state()
- term()

Variables

- parity

element

The element that may optionally be at the start of the state.

Parameters

```
element(value: str)
```

value `str`

The element itself.

This must not be empty.

fraction

A fraction, funnily enough.

Parameters

```
fraction(
  numerator: int str,
  denominator: int str
)
```

numerator `int` or `str`

The numerator itself.

Must conform to the rules of `number.value`.

denominator `int` or `str`

The denominator itself.

Must conform to the rules of `number.value`.

letter

A single character from the latin alphabet.

This is used primarily to differentiate between spectroscopic and jj notation when rendering.

Parameters

```
letter(value: str)
```

value `str`

The letter itself.

Must have `value.len() == 1`.

number

A single integer.

Parameters

```
number(value: int str)
```

value `int` or `str`

The number itself.

If a string then it must contain a valid integer.

orbital

A single orbital in the configuration list.

Parameters

```
orbital(
  principal: dictionary,
  azimuth: dictionary,
  occupation: dictionary,
  subscript: none dictionary
)
```

principal `dictionary`

The principle number of the orbital, n.

Accepted AST tags:

- `letter()`
- `number()`

azimuth `dictionary`

The azimuthal orbital number of the configuration list, l.

Accepted AST tags:

- `letter()`
- `number()`
- `fraction()`

occupation `dictionary`

The occupation number of the orbital.

Currently this should not be zero, this restriction may potentially be lifted in the future.

Accepted AST tags: `number()`

subscript `none` or `dictionary`

The subscript of the orbital.

Accepted AST tags:
- `number()`
- `fraction()`

Default: none

state

The full state of the atom.

Parameters

```
state(
  configuration: array,
  term: none dictionary,
  element: none dictionary
)
```

configuration `array`

The configuration list.

Each element must be an `orbital()` or a `term()`.

term `none` or `dictionary`

The total term of the state.

Accepted AST tags: `term()`

Default: none

element `none` or `dictionary`

The optional element at the start of the state.

Accepted AST tags: `element()`

Default: none

term

A term symbol.

Parameters

```
term(
  multiplicity: dictionary,
  character: dictionary,
  J: none dictionary,
  pi: int
)
```

multiplicity `dictionary`

The multiplicity of the term symbol, $2S + 1$.

Accepted AST tags: `number()`

character `dictionary`

The character of the term symbol.

Accepted AST tags:

- `letter()`
- `number()`
- `fraction()`

J `none` or `dictionary`

The J-value of the term symbol.

Accepted AST tags:

- `number()`
- `fraction()`

Default: `none`

pi `int`

The parity of the term symbol.

Must be equal to `parity.odd` or `parity.even`.

Default: `parity.even`

parity `dictionary`

The parity of a term symbol.

This dictionary contains only the fields `parity.odd` and `parity.even`.

This in, in practice, an enum, so the specific values of the fields should never be relied upon.