

# 2D Syntax

Version 8.11.1

November 27, 2023

```
#lang 2d      package: 2d-test
```

The `2d` language installs `#2d` reader support in the readtables, and then chains to the reader of another language that is specified immediately after `2d`.

The `#2d` syntax extension adds the ability use a two-dimensional grid syntax. That is, you can draw an ASCII-art grid and then treat that as an expression. For example, here is a simple equality function that operates on pairs and numbers, written using a `#2d` conditional expression:

```
#lang 2d racket
(require 2d/cond)

(define (same? a b)
  #2dcond
  |-----|
  |          | (pair? a) | (number? a) |
  |-----|
  | (pair? b) | (and (same? (car a) | #f |
  |          |           (car b)) |
  |          | (same? (cdr a) |
  |          |           (cdr b))) | |
|---|---|---|
  | (number? b) | #f | (= a b) |
  |-----|
  )
```

This notation works in two stages: reading, and parsing (just as in Racket in general). The reading stage converts anything that begins with `#2d` into a parenthesized expression (possibly signaling errors if the `≡` and `||` and `≠` characters do not line up in the right places).

Since the first line contains `#2dcond`, the reader will produce a sequence whose first position is the identifier `2dcond`.





evaluates to

```
'(2dex (10 10)
      (2 2)
      (((0 0)) 0)
      (((0 1)) 2)
      (((1 0)) 1)
      (((1 1)) 3))
```

and this

```
#lang 2d racket
```

```
'#2dex
```

0	1 2	3 4
5	6	

evaluates to

```
'(2dex (10 10 10)
      (2 2)
      (((0 0)) 0)
      (((0 1)) 5)
      (((1 0)) 1 2)
      (((1 1) (2 1)) 6)
      (((2 0)) 3 4))
```

In addition, the cells coordinates pairs have source locations of the first character that is inside the corresponding cell. (Currently the span is always 1, but that may change.)

## 1 Editing 2D

DrRacket provides a number of keybindings to help editing `#2d` expressions. See DrRacket's keyboard shortcuts.

## 2 2D Cond

```
(require 2d/cond)      package: 2d-lib

(2dcond cond-content)

cond-content = question-row
              body-row
              :
              | question-row
              body-row
              :
              else-row

question-row = empty-cell question-cell ...
              | empty-cell question-cell ... else-cell

body-row = question-cell exprs-cell ...

else-row = question-cell exprs-cell ... else-cell

question-cell = 

|               |
|---------------|
| question-expr |
|---------------|



empty-cell = 

|  |
|--|
|  |
|--|



exprs-cell = 

|               |
|---------------|
| expr expr ... |
|---------------|



else-cell = 

|      |
|------|
| else |
|------|


```

Evaluates the first row of question expressions until one of them returns a true value (signaling an error if none do), then evaluates the first column of question expressions until one of them returns a true value (signaling an error if none do), and then evaluates the cell in the middle where both point to, returning the result of the last expression in that cell.

### 3 2D Match

```
(require 2d/match)      package: 2d-lib

(2dmatch match-content)

  match-content = match-first-row
                  match-row
                  :
match-first-row = two-expr-cell match-pat-cell ...

                  match-row = match-pat-cell exprs-cell ...

two-expr-cell = [ [ col-expr row-expr ] ]

match-pat-cell = [ pat ]

exprs-cell = [ expr expr ... ]
```

Matches `col-expr` against each of patterns in the first column of the table and matches `row-expr` against each of the patterns in the row row, and then evaluates the corresponding `exprs-cell`, returning the value of the last expression in that cell.

Within the top-left cell, the leftmost expression will count as `col-expr`, and the rightmost as `row-expr`. In case of a tie (i.e., both expressions start at the same column, but on different lines), the bottommost one will count as `col-expr`. For example, all of these are valid:

```
[ [ col-expr row-expr ] ]
```

```
[ [ row-expr ] ]
[ col-expr ]
```

```
[ ]
```

`|| row-expr ||`  
`|| col-expr ||`  
`└──────────┘`

Changed in version 6.4 of package `2d-1ib`: Made scrutinee parsing more flexible.

## 4 2D Tabular

```
(require 2d/tabular)      package: 2d-lib

(2dtabular tabular-content)

tabular-content = tabular-row
                  :
                  | tabular-row
                  :
                  style-cell

tabular-row = tabular-cell ...

tabular-cell = tabular-expr ...

style-cell = style-content ...

style-content = #:style style-expr
                | #:sep sep-expr
                | #:ignore-first-row

style-expr : style?
sep-expr : (or/c block? content? #f)
tabular-expr : (or/c block? content?)
```

Constructs a `tabular` matching the given cells.

If a cell spans multiple columns, then the resulting `tabular` has `'cont` in the corresponding list element. No cells may span rows.

The `#:style` and `#:sep` arguments are just passed to `tabular`.

If the `#:ignore-first-row` keyword is provided, then the first row of the `2dtabular` expression is ignored. This can be used in case the first row of the rendered table should not have all of the columns (as `#2d` syntax requires that the first row contain a cell for each column that appears in the table).



## 5 2D Readtable

```
(require 2d/readtable)      package: 2d-lib
```

```
(make-readtable) → readtable?
```

Builds a `readtable?` that recognizes `#2d` and turns it into a parenthesized form as discussed in *2D Syntax*.

```
(2d-readtable-dispatch-proc char
                             port
                             source
                             line
                             column
                             position
                             /recursive
                             readtable) → any/c

char : char?
port : input-port?
source : any/c
line : (or/c exact-positive-integer? #f)
column : (or/c exact-nonnegative-integer? #f)
position : (or/c exact-positive-integer? #f)
/recursive : (-> input-port? any/c (or/c readtable? #f) any/c)
readtable : (or/c #f readtable?)
```

The function that implements `make-readtable`'s functionality. The `/recursive` function is used to handle the content in the cells.

See the docs on readtables for more information.

## 6 2d Lexer

```
(require 2d/lexer)      package: 2d-lib
```

```
(2d-lexer sub) → lexer/c  
sub : lexer/c
```

Constructs a `lexer/c` given one that handles lexing inside the cells.

