# Prodiams - Programmable Diagrams

The User Guide.
By Inspirel, rev. 1.1.

## Contents



# Introduction

Prodiams were created *out of frustration*.

Prodiams are intended to allow engineers to work with diagrams and schematics following the same standards and processes that are already established in the world of software engineering:

- Diagrams are specified by *small and human-readable programs*.
- Those prorgams can be *refactored* and *reused*, archived in *libraries* and integrated with other useful code.
- Diagrams are very easy to use in *version-control* and *change-control* systems, and they are diff- and patch-friendly. For the same reason, they are also easy to peer-review at the *source level*.

Prodiams were also intentionally made to be easy to generate or to act as sources for further processing, which makes them a versatile utility for model-based engineering and formal methods.

Prodiams is an *open framework* that can be extended with new layout rules and new shapes. For this reason, Prodiams is not supposed to be “complete” in any way - as such, it was meant to be in a state of continuous evolution.

## System requirements

Prodiams is a framework created in the Wolfram programming language and is intended to be used in the Mathematica environment. The smallest known system that allows to work with Prodiams with reasonable level of comfort is Raspberry Pi.

# "Hello World!"

This example is intentionally provided without any explanation:

```
<< Prodiams`

block[hw, "Hello World!"];
diagram[{hw}, {}]
```


# Basics

## Installation

As a software package, Prodiams is composed of (currently) three files:

- Prodiams.wl - the base package defining most fundamental functionality, not tied to any particular engineering domain.
- ProdiamsLogic.wl - extension of the base package with shapes for the most popular logic gates.
- ProdiamsElectronics.wl - extension of the base package with shapes of several basic electronic components.

These files are expected to grow as new shapes and rules are added to them and new extensions packages are likely to be created to cover other engineering disciplines and notations.

For most comfortable use, all these files should be put in one of the directories defined by the $Path variable.

## Loading the Prodiams packages

Prodiams packages are loaded as any other Mathematica package. Extension packages automatically pull the base package, so it is enough to load only the most specific package that is needed, for example:

```
<< ProdiamsLogic`
```

or:

```
<< ProdiamsElectronics`
```

From now on, all Prodiams definitions (from base package and from extension packages) are available and the above commands will not be repeated any more.

## Learning and getting help

This user guide itself is a Mathematica notebook and all examples in this guide are real - as such, they are immediately useful as a starting point for further modifications and learning by doing.

All Prodiams functions follow the standard Mathematica convention for documenting symbols and their descriptions can be retrieved in the usual way, for example:

```
?block
```

block[name,label,OptionsPattern[{width→5,height→5,style→None,comment→None}]] creates a simple block with a label.

```
?diagram
```

diagram[elements,layoutConstraints,OptionsPattern[{style→None}]]
draws a diagram including all elements and based on given layout constraints.
Optional style can override default settings defined in the defaultDiagramStyle association.

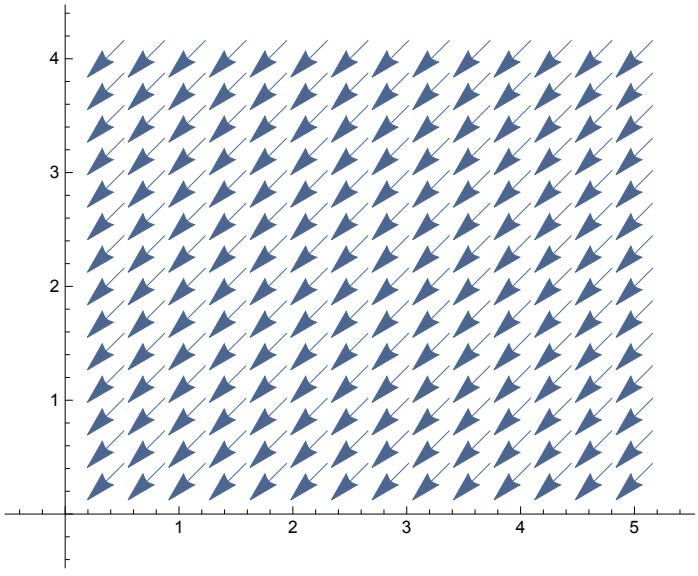
## Diagram's anatomy

Every diagram is composed of *shapes* (like logic gates or electronics components, connecting lines, etc.), that are placed in the 2-dimensional coordinate system according to some set of *layout constraints,* which are hints regarding the placement of shapes.

These two ingredients are defined separately and can be even separately refactored and reused, but their interaction is what allows the final diagram to be created. In order to make it easier for the user to define diagrams, there are also some implicit rules that are always in action.

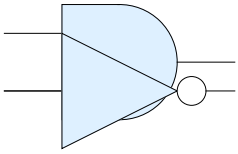
One of these implicit rules is that all shapes are drawn in the positive quarter of the coordinate system - that is, no shape is allowed to have negative coordinates, in both X and Y directions.

The other implicit rule is that there is a "gravity field" that attracts all shapes towards the origin of the coordinate system. This "gravity field" can be explained with the following plot:



The presence of this “gravity field”, with the implicit constraint that everything is kept on the positive side, means that if no rules are provided by the user, then all shapes will automatically fall to the bottom-left corner:

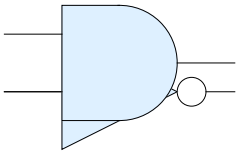
```
logicAND[a, {"A", "B"}, "C"];
logicNOT[n, "X", "Y"];

diagram[{a, n}, {}]
```



The “diagram” function is the main function creating diagrams and it accepts two arguments: the list of shapes to draw and the list of layout constraints to use. Above, the list of shapes contains two logic gates (“a” and “n”) and the list of layout constraints is empty, but thanks to the implicit rules described above, everything was attracted to the bottom-left corner.

The shapes are drawn in the order that they appear in the first list, which affects overlapping parts:

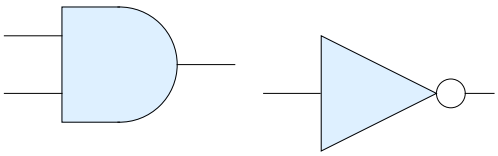
```
diagram[{n, a}, {}]
```



This can be a useful property in some types of diagrams.

In almost any realistic diagram, some layout constraints need to be added by the user to create meaningful visual structures:

```
diagram[{a, n}, {after[n, a]}]
```

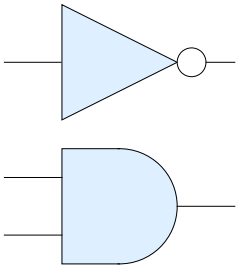


Above, the “after” function is the layout directive that provides constraints for two shapes - the “n” shape is placed after (on the right of) the “a” shape. Still, since there are no rules acting in the vertical direction, everything falls to the bottom thanks to the ever-active “gravity field”.

It is easy to see that the NOT gate is somewhat lower than the AND gate, but this can be explained by the fact that for these shapes, the layout is computed for the connectors only - now it is clear that the lower input of the AND gate is at the same level as both input and output of the NOT gate; these are really Y=0 coordinate values. Similarly, both input connectors of the AND gate are at X=0.

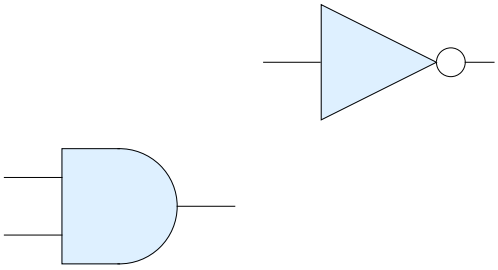
There are layout directives operating in the vertical direction, too:

```
diagram[{a, n}, {above[n, a]}]
```



Of course, now both shapes are attracted to the left (all three input connectors are at X=0). If one shape needs to be placed both on the right and above another shape, then this expectation needs to be expressed with appropriate combination of layout constraints:

```
diagram[{a, n},
 {
  after[n, a],
  above[n, a]
 }
]
```



There are many more layout directives to help establish relative positions between shapes - “before”, “below”, “align”, “middle”, “center” and their variants working on whole lists. Some of them are used in later examples in this guide and all are listed in the Reference chapter.

## Grid

In the diagrams above there was always a little bit of space left between shapes when “after” and “above” functions were used to establish their placement. This bit of space is not random and it is not fixed, either - but it is the first visible effect of another implicit constraint - that all coordinates are assigned natural values (no fractions).

This constraint can be visible here as well:

```
dot[d1]; dot[d2]; dot[d3]; dot[d4];
diagram[{d1, d2, d3, d4}, {allAfter[{d1, d2, d3, d4}]}]
```

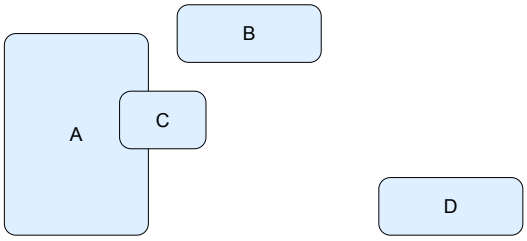
• • • •

The “allAfter” directive is a shorthand for a series of “after” directives applied for each consecutive pair. All four dots were placed on consecutive available “nodes” in the imaginary grid of natural coordinate values - that is, “d4” was placed at X=0, “d3” at X=1, “d2” at X=2 and “d1” and X=3 - and for the lack of any vertical constraint, all four dots have Y=0 due to the presence of “gravity” that pulls everything towards the origin.

The space that is left between shapes can be controlled with the “offset” option that is 1 by default,

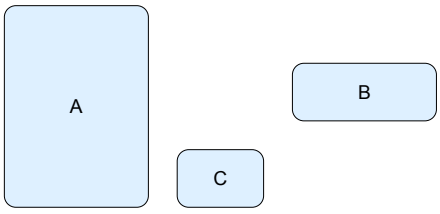
but can have other (integer!) values as well:

```
block[a, "A", height → 7];
block[b, "B", height → 2];
block[c, "C", width → 3, height → 2];
block[d, "D", height → 2];

diagram[{a, b, c, d},
 {
  allAbove[{b, c, d}],
  after[b, a],
  after[c, a, offset → -1],
  after[d, a, offset → 8]
 }
]
```




The important property of the constraints managed with these functions is that the offset (default or explicit) has the meaning of “at least”, not “exactly”. This “soft” property allows multiple constraints to cooperate:

```
diagram[{a, b, c},
 {
  above[b, c],
  after[b, a], (* <- here *)
  after[c, a],
  after[b, c] (* <- and here *)
 }
]
```

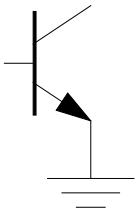



Above, directives marked by comments are not in conflict - “B” is after “A” *at least* 1 unit and it is also after “C” *at least* 1 unit. This cooperative property of layout directives allows easy modifications of the diagram when new shapes are inserted in between existing structures, which usually has only local impact and does not require the whole diagram to be redefined.

## Point-like vs. composite shapes

Some shapes, like the dot from previous examples, have very simple structure and can be fully described by a single coordinate pair - these are *point-like structures*. Other shapes can have more complex nature and can have multiple coordinate pairs for their different components. The following example shows the difference:

```
transistorNPN[t];
ground[g];

diagram[{t, g},
 {
  pin[connector[t, "E"], g]
 }
]
```

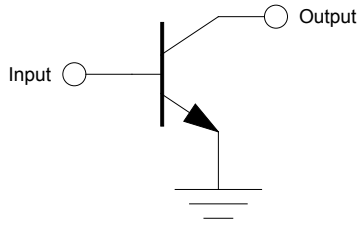


Above, there are two shapes - one represents the “ground” symbol, which is very simple, and another for the transistor symbol with three *connectors* for base, collector and emitter. The presence of these connectors is what makes the transistor a composite shape, with multiple coordinate pairs. Logic gates from previous examples are composite, too.

The "connector" function allows to create a reference for a single point of the given composite shape - as such, this function itself returns some point-like structure (called “anchor”), which can be further used to create layout constraints. The simplest layout constraint is “pin”, which just forces two point-like structures to be at the same location - here the ground symbol is pinned to the transistor’s emitter connector.

Of course, the other transistor’s connectors have names “B” and “C” - and all three can be used in layout constraints:

```
inPort[in, "Input"];
outPort[out, "Output"];

diagram[{t, in, out, g},
 {
  pin[connector[t, "B"], in],
  pin[connector[t, "C"], out],
  pin[connector[t, "E"], g]
 }
]
```

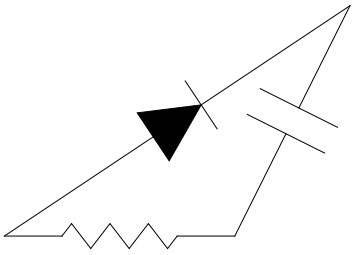



## Rigid vs. flexible shapes

Some composite shapes, like logic gates or transistors, have multiple points (connectors) with rigid structure between them. For example, the output connector of the logic NOT gate is always 8 grid units on the right of its input. These are *rigid shapes* as they define their own set of internal layout constraints.

There are other shapes, however, that are more flexible in nature and that allow the user to set up relations between their connectors. There are several two-connector electronics shapes that have this property:

```
resistor[r];
diode[d];
capacitor[c];

diagram[{r, d, c},
 {
  after[connector[r, "B"], connector[r, "A"], offset → 8],
  pin[connector[r, "B"], connector[c, "A"]],
  after[connector[c, "B"], connector[c, "A"], offset → 4],
  above[connector[c, "B"], connector[c, "A"], offset → 8],
  pin[connector[r, "A"], connector[d, "A"]],
  pin[connector[d, "B"], connector[c, "B"]]
 }
]
```



Such shapes can be stretched and placed quite freely (with some reasonable limitations - only the connector lines are flexible, the core shapes themselves are not), but at the same time they require a bit more careful coding, because they cannot take care of themselves.

The above example also demonstrates that two-connector electronic shapes all have connectors named “A” and “B” and that if there is any notion of “direction” (voltage and current sources, diodes, polarized capacitors, etc.), it flows from “A” to “B”. Other shapes have connectors with more domain-specific names (like transistor’s “B”, “C” and “E”) or are completely user-defined (like with all logic shapes).

# Intermediate - More Fun

## Labels

Most of the standard shapes support some form of labeling. For some of the shapes, like “inPort”, “outPort” and “block”, labels are essential and obligatory in the sense that they have to be provided as parameters when the shape is created:

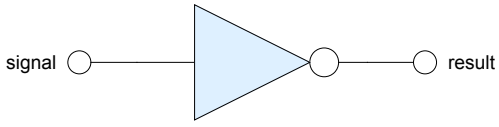
```
inPort[in, "signal"];
diagram[{in}, {}]
```


Other shapes do not have any labels by default, but support them in a uniform way. For example, logic gates have inputs and outputs with user-provided names:

```
logicNOT[n, "X", "Y"];
```

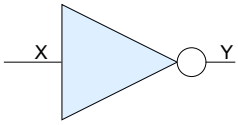
Above,”X” and “Y” are names that can be used to refer to input and output, respectively, for example in the “connector” function:

```
outPort[out, "result"];

diagram[{in, n, out},
 {
  pin[in, connector[n, "X"]],
  pin[out, connector[n, "Y"]]
 }
]
```



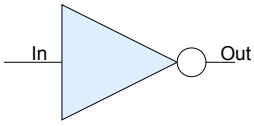

The connectors can have their own labels, though, and the easiest option is to automatically reuse connector names as labels:

```
logicNOT[n, "X", "Y", labels → Automatic];
diagram[{n}, {}]
```



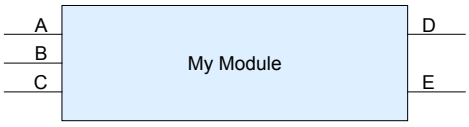
Connector labels do not have to be the same as their names:

```
logicNOT[n, "X", "Y", labels → {"X" → "In", "Y" → "Out"}];
diagram[{n}, {}]
```




The possibility to quickly label such shapes is especially convenient with shapes that have multiple inputs and outputs:

```
logicBlock[b, "My Module", {"A", "B", "C"},
  {"D", "E"}, connectorLabels → Automatic];
diagram[
 {b},
 {}]
```



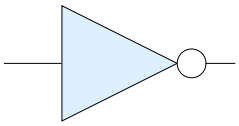

Shapes from the electronic family also support labels, but instead of labeling individual connectors (which usually have obvious meaning), there can be a single label per shape. The rules are similar as above:

```
resistor[r1]; (* no label *)
resistor[r2, label → Automatic]; (* reuse shape name *)
resistor[r3, label → "1.2kΩ"]; (* use arbitrary text *)

diagram[{r1, r2, r3},
 {
  after[connector[r1, "B"], connector[r1, "A"], offset → 8],
  pin[connector[r1, "B"], connector[r2, "A"]],
  after[connector[r2, "B"], connector[r2, "A"], offset → 8],
  pin[connector[r2, "B"], connector[r3, "A"]],
  after[connector[r3, "B"], connector[r3, "A"], offset → 8]
 }
]
```


## Comments

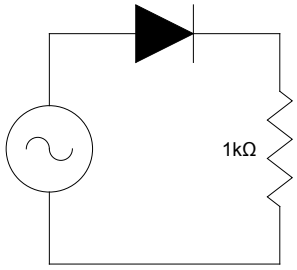
Independently from labels, shapes can have comments that are displayed as tooltips when the mouse is hovered over them (obviously, this feature fully works only in the Mathematica notebooks, not in PDF, and HTML has only limited support for this feature):

```
logicNOT[n, "X", "Y",
  comment →
   "This gate here is a hack, but the plane will not land without it."];
diagram[
 {n},
 {}]
```



Interestingly, comments are not limited to be text strings only - any Mathematica expression is fine, including plots and arbitrary images:

```
signalSource[ss, comment → Plot[3 Sin[t] Sin[10 t], {t, 0, 3 π}]];
diode[d, comment → Import[
     "http://www.learnabout-electronics.org/Semiconductors/images/diode-IV.gif"]
 ];
resistor[r, label → "1kΩ", comment → "This is some reasonable load."];
line[l, {"A", "B"}];

diagram[{ss, d, r, l},
 {
  above[connector[ss, "A"], connector[ss, "B"], offset → 8],
  pin[connector[ss, "A"], connector[d, "A"]],
  after[connector[d, "B"], connector[d, "A"], offset → 8],
  middle[connector[d, "B"], connector[d, "A"]],
  pin[connector[r, "A"], connector[d, "B"]],
  below[connector[r, "B"], connector[r, "A"], offset → 8],
  center[connector[r, "B"], connector[r, "A"]],
  pin[linePoint[l, "A"], connector[ss, "B"]],
  pin[linePoint[l, "B"], connector[r, "B"]]
 }
]
```




## Styles

Prodiams have several ways to influence styling settings like line or fill colors.

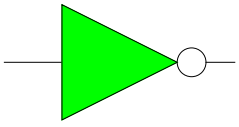
There is a global association variable, named “defaultDiagramStyle”, that provides common settings for all diagrams:

```
defaultDiagramStyle // Normal // Column
```

```
dotStyle → ■
dotRadius → 0.2
roundingRadius → 0.4
lineStyle → ■
connectorLineStyle → ■
fillStyle →
edgeStyle → ■
labelFunction → Function[{t, point, align}, {Black, Text[t, point, align]}]
gridSize → 15
diagramPadding → 4
printStats → False
```

It is possible to change these values globally, which will affect all diagrams created from that point in time:

```
defaultDiagramStyle[fillStyle] = Green;
diagram[{n}, {}]
```

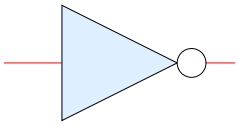


But for the sake of having stable foundations for the rest of this user guide, this setting will be reverted to its default value:

```
defaultDiagramStyle[fillStyle] = LightBlue;
```

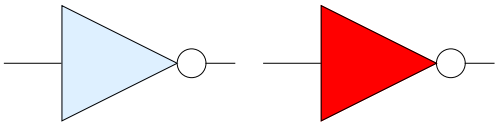
It is possible to override some (or all) of the settings with optional “style” setting in the diagram function and this affects only the given diagram:

```
diagram[{n}, {}, style → {connectorLineStyle → Red}]
```



It is also possible to provide such styling options when the shape is created - this way, the shape-specific settings will be used in all diagrams where that shape is used:

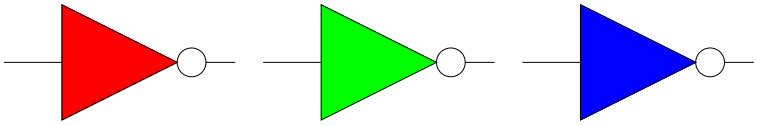
```
logicNOT[n1, "In", "Out"];
logicNOT[n2, "In", "Out", style → {fillStyle → Red}];

diagram[{n1, n2}, {after[n2, n1]}]
```



Above, "n2" has its own style, which is merged with default style settings whenever that shape is used. Such shape-specific styles have priority over diagram-specific styles (which themselves override default global styles).

As an ultimate way to override all other settings, the “styled” wrapper allows to define style options that override everything else. The following example demonstrates the complete hierarchy of style settings:

```
logicNOT[n1, "In", "Out"];
logicNOT[n2, "In", "Out", style → {fillStyle → Green}];
logicNOT[n3, "In", "Out", style → {fillStyle → Green}];

diagram[{n1, n2, styled[n3, style → {fillStyle → Blue}]},
 {allAfter[{n3, n2, n1}]},
 style → {fillStyle → Red}
]
```



In the above example:

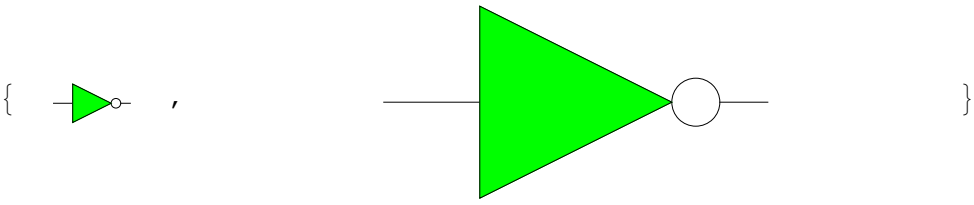
- "n1" would have been "LightBlue" (due to default settings), but the diagram-level settings override it to "Red",
- "n2" is “Green”, because it has its shape-specific style, which is more important than the diagram-level setting,
- "n3" has the same specification as “n2”, but is “Blue” instead, because it was used within the “styled” wrapper that overrides all other settings.

Some of the style settings deserve additional explanation.

The "labelFunction” value in the style association denotes a function that can accept the same format of arguments as the standard Text function (the variant with offset arguments). In the simplest case it might be just set to Text, but there will be no control over font or its decoration. The anonymous function in the default setting shows a possible way to wrap the standard Text function with additional style directives. Of course, such function can be also defined separately, and - last but not least - actual mechanisms can be arbitrary and the provided function might do something else than just printing text.

The “gridSize” setting defines the size, in pixels in the final image, of the single grid unit. Changing this value has the effect of resizing all shapes:

```
{diagram[{n2}, {}, style → {gridSize → 5}],
 diagram[{n2}, {}, style → {gridSize → 25}]}
```



The “diagramPadding” is an amount of white space, in grid units, that is artificially added around the whole diagram.

Finally, “printStats”, when set to True, allows to measure and print some performance statistics - this setting is useful only for debugging or for identifying performance optimization strategies with large diagrams.

## Wrappers

Wrappers are functions that temporarily change some properties of their arguments - one such wrapper is the “styled” function from previous examples, that allows to locally override any style setting.

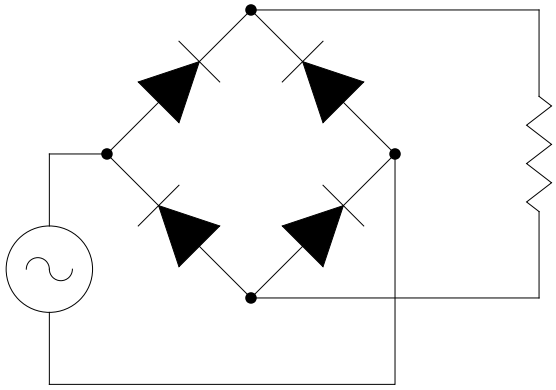
Another useful wrapper is the “hidden” function that suppresses the drawing of the given shape (or a list of shapes), while still using them for finding the layout solution - that is, the given shape still takes its space in the diagram, but is invisible, which allows to create several variants of the same diagram, showing different aspects of the same problem.

The following example can help demonstrate this concept:

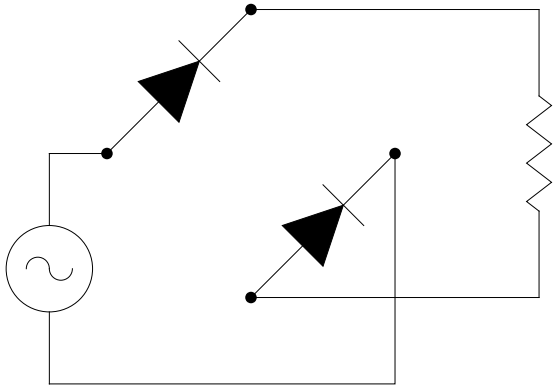
```
signalSource[ss];
diode[d1];
```

```
diode[d2];
diode[d3];
diode[d4];
resistor[r];
line[l1, {"A", "B"}];
line[l2, {"A", "B", "C"}];
line[l3, {"A", "B"}];
line[l4, {"A", "B"}];
dot[dot1];
dot[dot2];
dot[dot3];
dot[dot4];

layout = {
    above[connector[ss, "A"], connector[ss, "B"], offset → 8],
    pin[connector[ss, "A"], linePoint[l1, "A"]],
    after[linePoint[l1, "B"], linePoint[l1, "A"], offset → 2],
    middle[linePoint[l1, "B"], linePoint[l1, "A"]],
    pin[linePoint[l1, "B"], dot1],
    pin[dot1, connector[d1, "A"]],
    above[connector[d1, "B"], connector[d1, "A"], offset → 5],
    after[connector[d1, "B"], connector[d1, "A"], offset → 5],
    pin[connector[d1, "B"], dot2],
    pin[dot2, connector[d2, "B"]],
    middle[connector[d2, "A"], connector[d1, "A"]],
    after[connector[d2, "A"], connector[d2, "B"], offset → 5],
    pin[connector[d2, "A"], dot3],
    pin[dot1, connector[d3, "B"]],
    after[connector[d3, "A"], connector[d3, "B"], offset → 5],
    above[connector[d3, "A"], connector[d3, "B"], offset → -5],
    pin[dot4, connector[d3, "A"]],
    pin[dot4, connector[d4, "A"]],
    pin[dot3, connector[d4, "B"]],
    pin[connector[ss, "B"], linePoint[l2, "A"]],
    center[linePoint[l2, "B"], dot3],
    pin[linePoint[l2, "C"], dot3],
    pin[dot2, linePoint[l3, "A"]],
    after[linePoint[l3, "B"], linePoint[l3, "A"], offset → 10],
    middle[linePoint[l3, "B"], linePoint[l3, "A"]],
    pin[dot4, linePoint[l4, "A"]],
    center[linePoint[l4, "B"], linePoint[l3, "B"]],
    middle[linePoint[l4, "B"], linePoint[l4, "A"]],
    pin[linePoint[l3, "B"], connector[r, "A"]],
    pin[linePoint[l4, "B"], connector[r, "B"]]
  };
```

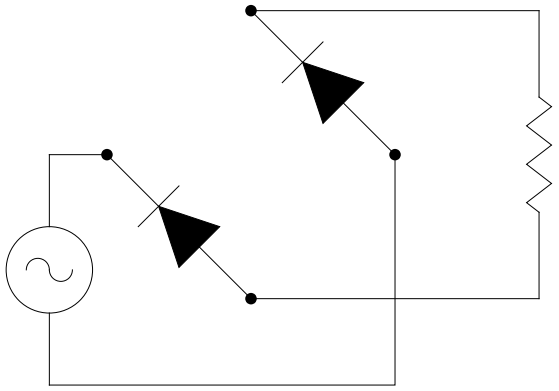
```
complete =
 diagram[{ss, d1, d2, d3, d4, r, dot1, dot2, dot3, dot4, l1, l2, l3, l4}, layout]
```



```
positive = diagram[
   {ss, d1, d4, hidden[{d2, d3}], r, dot1, dot2, dot3, dot4, l1, l2, l3, l4}, layout]
```

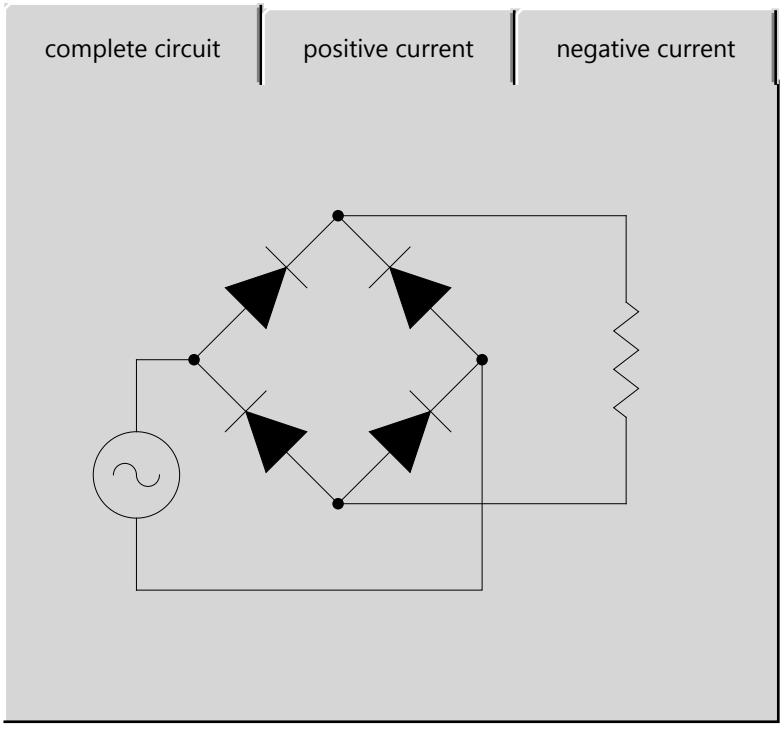


```
negative = diagram[
  {ss, hidden[{d1, d4}], d2, d3, r, dot1, dot2, dot3, dot4, l1, l2, l3, l4}, layout]
```



The “complete”, “positive” and “negative” are three diagrams that have exactly the same layout, but differ in the visibility of some of the shapes. Now it is easy to create a nice-looking viewer for all three variants:

```
TabView[{
  "complete circuit" → complete,
  "positive current" → positive,
  "negative current" → negative
 }]
```



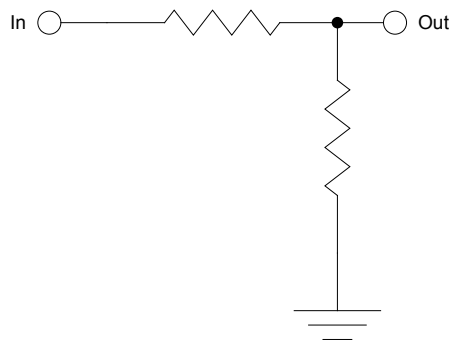
The “styled” wrapper, explained above in the subsection devoted to style options, can also be used to achieve the effect of highlighting some aspects of the diagram like signal paths or important sub-modules.

## Diagram refactoring

Some possibilities to refactor diagram definitions were already used above, where the longish sequence of layout constraints was assigned a separate name and that name was reused with three different diagrams - this definitely reduced the amount of code that would have to be written (or, worse, copy-pasted) otherwise. It would be also possible to refactor only some part of the sequence, or do similar simplifications in the list of shapes as well.

Such refactorings are very easy to do, but even more interesting effects can be achieved with parameterized functions.

```
inPort[in, "In"];
outPort[out, "Out"];
resistor[r1];
resistor[r2];
dot[d12];
ground[g];

diagram[{in, r1, r2, d12, g, out},
 {
  pin[in, connector[r1, "A"]],
  after[connector[r1, "B"], connector[r1, "A"], offset → 8],
  middle[connector[r1, "B"], connector[r1, "A"]],
  pin[connector[r1, "B"], d12],
  pin[connector[r2, "A"], d12],
  above[connector[r2, "A"], connector[r2, "B"], offset → 8],
  center[connector[r2, "A"], connector[r2, "B"]],
  pin[connector[r2, "B"], g],
  pin[d12, out]
 }
]
```

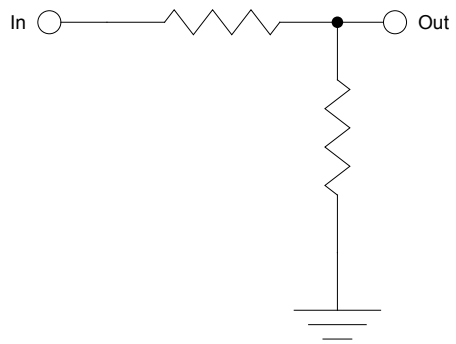



The above diagram involves a common “impedance divider” pattern used in many different circuits and it makes sense to refactor this common pattern into a separate function:

```
impDivider[in_, out_, common_, e1_, e2_] :=
 {
  pin[in, connector[e1, "A"]],
  after[connector[e1, "B"], connector[e1, "A"], offset → 8],
  middle[connector[e1, "B"], connector[e1, "A"]],
  pin[connector[e1, "B"], out],
  pin[connector[e2, "A"], out],
  above[connector[e2, "A"], connector[e2, "B"], offset → 8],
  center[connector[e2, "A"], connector[e2, "B"]],
  pin[connector[e2, "B"], common]
 }
```

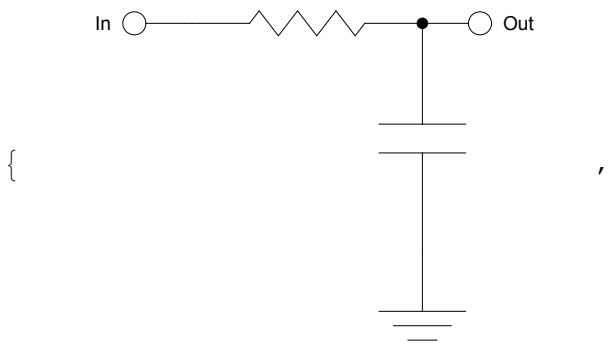
The “impDivider” function abstracts away the common design pattern and resolves to proper list of layout directives applied to shapes that are given as its parameters. Now the diagram definition can be much shorter:

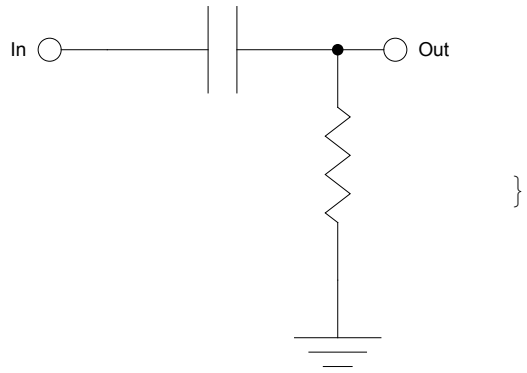
```
diagram[{in, r1, r2, d12, g, out},
 {
  impDivider[in, d12, g, r1, r2],
  pin[d12, out]
 }
]
```




But a real benefit from this refactorization is that the same design pattern can be reused in other diagrams as well:

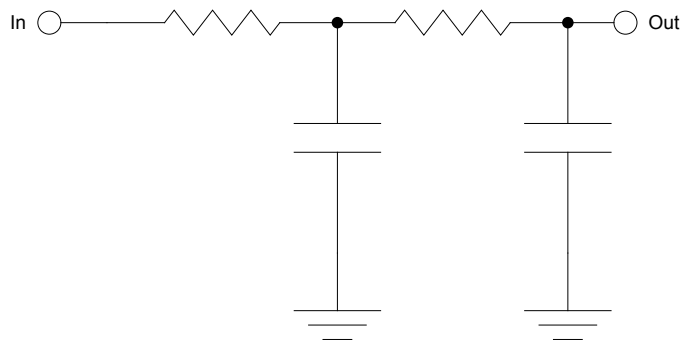
```
capacitor[c1];

lowPass = diagram[{in, r1, c1, d12, g, out},
    {
     impDivider[in, d12, g, r1, c1],
     pin[d12, out]
    }
  ];
highPass = diagram[{in, c1, r2, d12, g, out},
    {
     impDivider[in, d12, g, c1, r2],
     pin[d12, out]
    }
  ];

{lowPass, highPass}
```




Or it can be even used multiple times within the same diagram:

```
capacitor[c2];
dot[d22];
ground[g2];

diagram[{in, r1, c1, d12, g, r2, c2, d22, g2, out},
 {
  impDivider[in, d12, g, r1, c1],
  impDivider[d12, d22, g2, r2, c2],
  pin[d22, out]
 }
]
```




Most importantly, such functions can be stored in a separate library or package and used as a common set of utilities - just as would be done in any reasonably managed project.

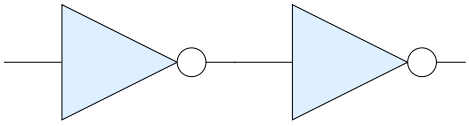
## Semantics

The short version is: there is no semantics.

Prodiams allows to draw diagrams without any respect to whether they make sense or not. As such, *it is just a drawing tool*.

Interestingly, it is also possible to create diagrams that *look* reasonably well, but that do not convey the intended function:

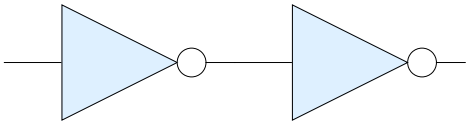
```
logicNOT[n1, "X", "Y"];
logicNOT[n2, "X", "Y"];

diagram[{n1, n2},
 {after[connector[n2, "X"], connector[n1, "Y"], offset → 0]}
]
```



Above, the two gates *look like* they are connected, but this is only a visual coincidence - the shapes are placed somewhere on the diagram and incidentally the connectors seem to be matched, but if that was the intention of the designer, it was not conveyed in the source code.

A much better way to express the design intent is to express it with “pin” directives:

```
diagram[{n1, n2},
 {pin[connector[n2, "X"], connector[n1, "Y"]]}
]
```



This diagram looks the same as the previous one, but the sequence of layout directives contains information that can be extracted by otheer means and further processed or analyzed. In a bigger diagram, the set of “pin” instances can be sufficient to extract the complete connectivity map. It is recommended to follow this as a matter of good engineering practice.

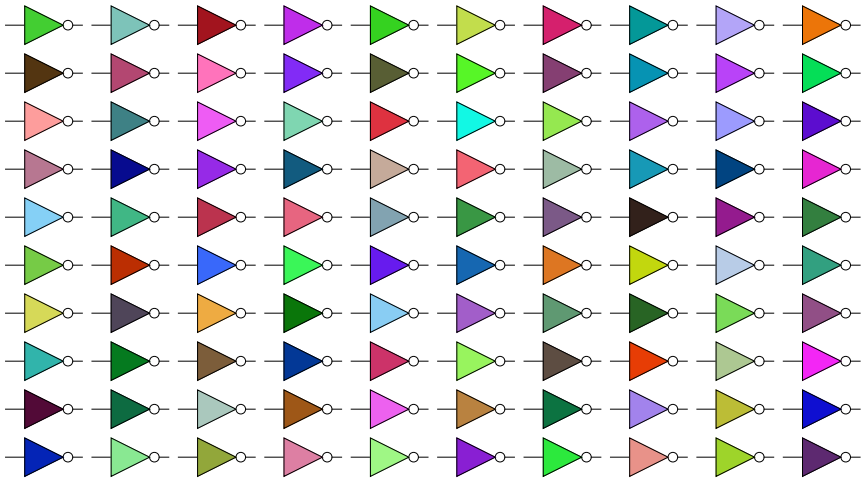
## Integration with Mathematica

Many of the examples in this guide already reached outside of the Prodiams package. Whether these are style settings referring to standard Mathematica color names or the possibility to create helper functions to refactor parts of the diagram - Prodiams work within the context of a much more versatile computing platform. Actually, considering the vast number of Mathematica functions, it would be difficult to name the limits of how much can be done.

The following example is definitely not meant to exhaust the subject, but hints at the possibility to

write programs that generate diagrams:

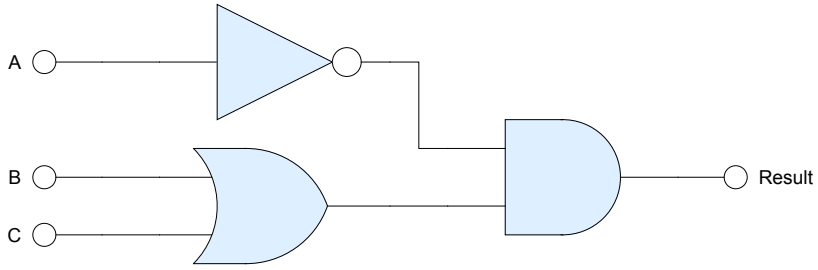
```
nots = Table[
    logicNOT[Evaluate[Unique["not"]], "X", "Y",
     style → {fillStyle → RandomColor[]}],
    {i, 100}];

rows = Partition[nots, 10];

diagram[nots,
 {
  Map[allBefore, rows],
  Map[
   Function[row,
    Map[
     Function[neighbours,
      Apply[middle, neighbours]
     ],
     Partition[row, 2, 1]
    ]
   ],
   rows
  ],
  Map[
   Function[neighbourRows, Apply[above, neighbourRows[[All, 1]]]],
   Partition[rows, 2, 1]
  ]
 },
 style → {gridSize → 5}
]
```



The above diagram probably does not represent any reasonable system design, but is cool anyway and has high “management convincing potential”.

A more convincing example is included in the ProdiamsLogic package itself, in the form of the logicDiagram function that attempts to draw a reasonable diagram from the given logic expression:

```
logicDiagram[¬ A ⋀ (B ⋁ C)]
```




Internally, the logicDiagram function decomposes the given expression and generates the list of shapes, together with appropriate layout constraints.

# Advanced - Under the Hood

This chapter is intended for advanced users and explains the exact steps that are taken by the main engine in order to create the diagram. This description can be useful for those who would like to improve the framework or create new shapes.

The following, very simple diagram is dissected in this chapter:

```
inPort[in, "input"];
resistor[r];
outPort[out, "output"];

diagram[{in, r, out},
 {
  pin[in, connector[r, "A"]],
  after[connector[r, "B"], connector[r, "A"], offset → 8],
  pin[out, connector[r, "B"]]
 }
]
```


As was already explained in earlier chapter, without user-provided layout constraint, everything would fall to the origin of the coordinate system due to the “gravity field” that is always active. In this

context, user-provided constraints are additional equalities and inequalities that are added to the set of constraints and the solution is found for all involved coordinates.

The main engine learns about all involved coordinates by calling the “internalCoordinates” on all shapes:

```
internalCoordinates[in]
```

```
{in[x], in[y]}
```

The following shows the exact nature of what the “in” shape contributes:

```
internalCoordinates[in] // InputForm
```

```
{"in"["x"], "in"["y"]}
```

These are two expressions that are very readable for a human (even though this readability is not needed other than for debugging newly developed shapes), but from the Mathematica perspective they are not resolved and can be used as variable names. Intuitively, the “in” shape, being a point-like structure, contributes just two variables for its coordinates.

The output port “out” has similar set of coordinates, as it is also a point-like structure:

```
internalCoordinates[out]
```

```
{out[x], out[y]}
```

The resistor is a complex shape with two connectors and its set of coordinates reflects this:

```
internalCoordinates[r]
```

```
{r[A][x], r[A][y], r[B][x], r[B][y]}
```

Again, the “structure” of these expressions is irrelevant, although the “connector” function follows the same conventions:

```
connector[r, "A"]
```

```
anchorT[r[A][x], r[A][y]]
```

```
connector[r, "B"]
```

```
anchorT[r[B][x], r[B][y]]
```

The “connector” function creates an anchor object, which is an invisible point-like structure, referring to the same coordinates that the resistor reports on the list returned from “internalCoordinates”. It is therefore recommended to follow the same naming convention in newly developed shapes.

In any case, there are 8 variables involved in the whole diagram. Without any user-provided layout constraints, all these variables would be assigned 0 as a layout solution, which will be explained later on.

The layout constraints introduce relations between these variables. The “pin” directive is actually just a function that creates equalities for the coordinates of its arguments:

```
pin[in, connector[r, "A"]]
```

```
{in[x] == r[A][x], in[y] == r[A][y]}
```

Similarly:

```
pin[out, connector[r, "B"]]
```

```
{out[x] == r[B][x], out[y] == r[B][y]}
```

The “after” constraint is contributing some “stretch” to the whole structure:

```
after[connector[r, "B"], connector[r, "A"], offset → 8]
```

```
{r[B][x] ≥ 8 + r[A][x]}
```

These constraints express, in terms of coordinate variables, the relative positions of shapes, as intended in the use of the three directives - all layout directives just operate on coordinate variables, creating different equalities and inequalities.

What is important here is that layout constraints can be contributed not only by the user, but also by shapes themselves. Shapes communicate their own constraints by means of the “internalLayout” function - in this example, there are no such constraints:

```
{internalLayout[in], internalLayout[r], internalLayout[out]}
```

```
{{}, {}, {}}
```

The reason for this is that the shapes that are used in this diagram are either point-like structures (“in”, “out” - with just one coordinate pair there is no relation to contribute) or a flexible shape (resistor), which is set up by means of user-provided constraints. But there are also rigid shapes that have their own internal structure and can contribute their own constraints, for example:

```
logicNOT[n, "In", "Out"];
internalLayout[n]
```

```
{8 + n[In][x] == n[Out][x], n[In][y] == n[Out][y]}
```

```
transistorNPN[t];
internalLayout[t]
```

```
{t[C][x] == 3 + t[B][x], t[C][y] == 2 + t[B][y],
 t[E][x] == 3 + t[B][x], t[E][y] == -2 + t[B][y]}
```

Ultimately, it does not really matter where the constraints come from - both internal and user-provided constraints are merged together to create a single set of equalities and inequalities. This set is further extended by adding global constraints that keep everything in the positive quarter of the coordinate system:

```
allCoordinates = Flatten[Map[internalCoordinates, {in, r, out}]]
```

```
{in[x], in[y], r[A][x], r[A][y], r[B][x], r[B][y], out[x], out[y]}
```

```
allPositive = Map[Function[v, v ≥ 0], allCoordinates]
```

```
{in[x] ≥ 0, in[y] ≥ 0, r[A][x] ≥ 0, r[A][y] ≥ 0,
 r[B][x] ≥ 0, r[B][y] ≥ 0, out[x] ≥ 0, out[y] ≥ 0}
```

This set of equalities and inequalities forms a solution space, where the minimum of the sum of all coordinates is found. The minimum of the sum of all coordinates is what creates the “gravity field” that pulls all variables towards the origin of the coordinate system:

```
Total[allCoordinates]
```

```
in[x] + in[y] + out[x] + out[y] + r[A][x] + r[A][y] + r[B][x] + r[B][y]
```

It should be now obvious why, without any user-provided constraints, all shapes would fall to the origin of the coordinate system - this is where the above function has its minimum. But user-provided constraints, which usually affect relative placement of shapes, expand the solution further apart.

The actual workhorse of the Prodiams package, and the central point of the “diagram” function, is the Minimize function, which is called like this (taking into account above expressions):

```
solution = Minimize[
  {
   Total[allCoordinates], (* the goal function *)
   {
    (* list of all constraints, internal and user-provided: *)
    pin[in, connector[r, "A"]],
    after[connector[r, "B"], connector[r, "A"], offset → 8],
    pin[out, connector[r, "B"]],

    (* added as a global constraint: *)
    allPositive
   }
  },
  allCoordinates, (* what is being solved *)
  Integers (* the solution space *)
 ]
```

```
{16, {in[x] → 0, in[y] → 0, r[A][x] → 0,
  r[A][y] → 0, r[B][x] → 8, r[B][y] → 0, out[x] → 8, out[y] → 0}}
```

The first value above is the value of the goal function, which is discarded (who cares what is the sum of coordinates?) and the second part is the actual solution. This solution is then fed to the “draw” function that is called for each shape, which also accepts the hierarchically merged style settings. As a funny example, the following is the list of graphics primitives that “draw” returns for the resistor in this diagram:

```
draw[r, Association[solution[[2]]], defaultDiagramStyle]
```

$$\left\{\blacksquare, \text{Line}\left[\left\{\{2, 0\}, \left\{\frac{7}{3}, \frac{\sqrt{3}}{4}\right\}, \left\{3, -\frac{\sqrt{3}}{4}\right\}, \left\{\frac{11}{3}, \frac{\sqrt{3}}{4}\right\}, \left\{\frac{13}{3}, -\frac{\sqrt{3}}{4}\right\}, \left\{5, \frac{\sqrt{3}}{4}\right\}, \left\{\frac{17}{3}, -\frac{\sqrt{3}}{4}\right\}, \{6, 0\}\right\}\right], \blacksquare, \text{Line}[\{\{0, 0\}, \{2, 0\}\}], \text{Line}[\{\{6, 0\}, \{8, 0\}\}]\right\}$$

As can be seen from the last elements in this list, the resistor shape in this diagram extends from {0,0} to {8,0}, which corresponds to the solution for its coordinate variables.

All such contributions, from all involved shapes, are combined and the final graphics is created:

```
Graphics[Flatten[
  Map[
   Function[e, draw[e, Association[solution[[2]]], defaultDiagramStyle]],
   {in, r, out}
  ]
 ]]
```


As a final touch, the main engine checks the “boundingBox” of all involved shapes, which allows to determine the extent of the whole diagram, so that proper padding and plot range can be selected. Image size is determined from “gridSize” style setting and the extent of the complete diagram.

That’s it!

The above explanation can be helpful for those who would like to extend the framework or develop new shapes - in which case the important take-away from the above is that each new shape needs to implement the following functions:

- “internalCoordinates” - returns the list of coordinate variables for the given shape; the existing naming convention should be preserved to benefit from some of the existing helper functions,
- “internalLayout” - returns the shape’s contribution to the list of coordinate constraints; this list can be empty for point-like structures and flexible shapes that rely on user-provided constraints for their layout,
- “specificStyle” (not used above) - returns the set of shape-specific style options; these values are merged with default and diagram-specific settings,
- “boundingBox” - returns the Rectangle-like coordinates (bottom-left - top-right) of the shape’s extent,
- “draw” - returns the list of graphics directives and primitives for the given shape, layout solution, and style settings.

Of course, the existing shapes can be used as a reference and a source of code patterns to reuse.

# Reference

## List of layout directives

All layout directives are defined in the base Prodiams package.

pin[p1,p2] is a layout directive, forces two point–like structures to be in the same place.

before[e1,e2,OptionsPattern[{offset→1}]] is a layout directive,
puts e1 on the left of e2, with at least offset distance between their bounding boxes.

allBefore[es,OptionsPattern[{offset→1}]] is a layout directive, accepts a list of elements and puts every element from the list on the left of its successor, with at least offset distance between their bounding boxes.
allBefore[es1,es2] accepts two lists and ensures that all elements from es1 are on the left of all elements from es2.

after[e1,e2,OptionsPattern[{offset→1}]] is a layout directive, puts e1 on the right of e2, with at least offset distance between their bounding boxes.

allAfter[es,OptionsPattern[{offset→1}]] is a layout directive, accepts a list of elements and puts every element from the list on the right of its successor, with at least offset distance between their bounding boxes.
allAfter[es1,es2] accepts two lists and ensures that all elements from es1 are on the right of all elements from es2.

above[e1,e2,OptionsPattern[{offset→1}]] is a layout directive, puts e1 above e2, with at least offset distance between their bounding boxes.

allAbove[es,OptionsPattern[{offset→1}]] is a layout directive, accepts a list of elements and puts every element from the list above its successor, with at least offset distance between their bounding boxes.
allAbove[es1,es2] accepts two lists and ensures that all elements from es1 are above all elements from es2.

below[e1,e2,OptionsPattern[{offset→1}]] is a layout directive, puts e1 below e2, with at least offset distance between their bounding boxes.

allBelow[es,OptionsPattern[{offset→1}]] is a layout directive, accepts a list of elements and puts every element from the list below its successor, with at least offset distance between their bounding boxes.
allBelow[es1,es2] accepts two lists and ensures that all elements from es1 are below all elements from es2.

alignLeft[e1,e2,OptionsPattern[{offset→0}]] is a layout directive, left−aligns the bounding boxes of its arguments, with optional offset.

allAlignLeft[es] is a layout directive, accepts a list of elements and left−aligns all their bounding boxes.

alignRight[e1,e2,OptionsPattern[{offset→0}]] is a layout directive, right−aligns the bounding boxes of its arguments, with optional offset.

allAlignRight[es] is a layout directive, accepts a list of elements and right−aligns all their bounding boxes.

center[e1,e2] is a layout directive, centers horizontally the bounding boxes of its arguments.

allCenter[es] is a layout directive, accepts a list of elements and centers horizontally all their bounding boxes.

alignTop[e1,e2,OptionsPattern[{offset→0}]] is a layout directive, top−aligns the bounding boxes of its arguments, with optional offset.

allAlignTop[es] is a layout directive, accepts a list of elements and top−aligns all their bounding boxes.

alignBottom[e1,e2,OptionsPattern[{offset→0}]] is a layout
directive, bottom−aligns the bounding boxes of its arguments, with optional offset.

allAlignBottom[es] is a layout directive, accepts a list of elements and bottom−aligns all their bounding boxes.

middle[e1,e2] is a layout directive, centers vertically the bounding boxes of its arguments.

allMiddle[es] is a layout directive, accepts a list of elements and centers vertically all their bounding boxes.

## List of wrappers

All wrappers are defined in the base Prodiams package.

styled[e,OptionsPattern[{style−>None}]] is a wrapper that allows to override all style settings for the given element.

hidden[e] is a wrapper that prevents the given element from being displayed (but it still takes part in layout solution).

## List of shapes

### Prodiams (base package)

anchor[name] creates a point−like structure that has no visual
representation (it is an invisible point) and is typically used to connect elements together.
Anchor objects are also used as intermediate values when connector points are referred.

extent[p1,p2] creates an invisible box that spans two point−like structures.
Extents can be used as helpers for centering shapes between other, more distant objects.

dot[name,OptionsPattern[{style→None,comment−>None}]] creates a point−like structure, displayed as a small disk.

block[name,label,OptionsPattern[{width→5,height→5,style→None,comment→None}]] creates a simple block with a label.

line[name,pointNames,OptionsPattern[{style→None,comment→None}]]
creates a multi−segment line with named ends and intermediate points.

linePoint[l,pointName] creates an anchor for the given named line point.

inPort[name,label,OptionsPattern[{style→None,comment→None}]] creates an input point with a label.

outPort[name,label,OptionsPattern[{style→None,comment→None}]] creates an output point with a label.

connector[e,conName] creates an anchor for the given element's connection point.

## ProdiamsElectronics (extension package)

ground[name,OptionsPattern[{style–>None,comment–>None}]] creates
a ground symbol, which itself is a point–like structure without any explicit connectors.

resistor[name,OptionsPattern[{label→None,style→None,comment→None}]] creates a resistor symbol.
This shape has connectors named A and B.

capacitor[name,OptionsPattern[{polarized→False,label→None,style→None,comment→None}]]
creates a capacitor symbol for both polarized and non–polarized variants.
This shape has connectors named A and B.

diode[name,OptionsPattern[{zener→False,label→None,style→None,comment→None}]]
creates a diode symbol with optional Zener marks.
This shape has connectors named A and B.

zenerDiode[name,OptionsPattern[{label→None,style→None,comment→None}]]
convenience wrapper for the diode function, creates a Zener diode symbol.
This shape has connectors named A and B.

voltageSource[name,OptionsPattern[{label→None,style→None,comment→None}]] creates a voltage source symbol.
This shape has connectors named A and B.

currentSource[name,OptionsPattern[{label→None,style→None,comment→None}]] creates a current voltage symbol.
This shape has connectors named A and B.

signalSource[name,OptionsPattern[{label→None,style→None,comment→None}]] creates a generic signal source symbol.
This shape has connectors named A and B.

bipolarTransistor[name,type,OptionsPattern[{orientation→Right,rotation→0,label→None,style→None,comment→None}]]
creates a bipolar transistor symbol of the given type (1–NPN, 2–PNP),
orientation (Left/Right) and rotation, expressed in the number (0–3) of 90° units.
This shape has connectors named B, C and E.

transistorNPN[name,OptionsPattern[{orientation→Right,rotation→0,label→None,style→None,comment→None}]] creates an
NPN transistor symbol of the given orientation (Left/Right) and rotation, expressed in the number (0–3) of 90° units.
This shape has connectors named B, C and E.

transistorPNP[name,OptionsPattern[{orientation→Right,rotation→0,label→None,style→None,comment→None}]] creates a
PNP transistor symbol of the given orientation (Left/Right) and rotation, expressed in the number (0–3) of 90° units.
This shape has connectors named B, C and E.

## ProdiamsLogic (extension package)

logicNOT[name,inName,outName,OptionsPattern[{labels→None,style→None,comment→None}]]
creates a NOT gate with named connectors and optional labels.

logicAND[name,inNames,outName,OptionsPattern[{labels→None,outputNegated→False,style→None,comment→None}]]
creates an AND gate with named connectors and optional labels.

logicNAND[name,inNames,outName,OptionsPattern[{labels→None,style→None,comment→None}]]
creates a NAND gate with named connectors and optional labels.

logicOR[name,inNames,outName,OptionsPattern[{labels→None,outputNegated→False,inputsExclusive→False,style→None,
comment→None}]] creates an OR gate with named connectors and optional labels.

logicNOR[name,inNames,outName,OptionsPattern[{labels→None,style→None,comment→None}]]
creates a NOR gate with named connectors and optional labels.

logicXOR[name,inNames,outName,OptionsPattern[{labels→None,style→None,comment→None}]]
creates a XOR gate with named connectors and optional labels.

logicXNOR[name,inNames,outName,OptionsPattern[{labels→None,style→None,comment→None}]]
creates a XNOR gate with named connectors and optional labels.

logicBlock[name,label,inNames,outNames,OptionsPattern[{width→16,connectorLabels→Automatic,style→None,comment→
None}]] creates a generic block with the given label, number of inputs and outputs.

logicDiagram[expr_,OptionsPattern[{style→None,debug→False}]]
creates a best−effort diagram from the given logical expression.

## Shape API

Shape API is a set of functions that allow interactions between main diagrams engine and (newly developed) shapes.

Note: all these functions are first used in the base Prodiams package and their implementations for new shapes need to be kept in the base context, even if shapes themselves are implemented in other (extension) packages.

internalCoordinates[e] returns a list of internal coordinate names for the given element.
This function needs to be implemented (overridden) for new element types.

internalLayout[e] returns a list of constraints (equalities or inequalities) involving internal coordinates of the given element.
This function needs to be implemented (overridden) for new element types.

boundingBox[e] returns a Rectangle–like coordinates of the smallest
imaginary box that contains the given element. The meaning of this is not strict.
This function needs to be implemented (overridden) for new element types.

specificStyle[e] returns an association or a (possibly
empty) list of rules that override default– and diagram–level style values.
This function needs to be implemented (overridden) for new element types.

draw[e,layout,style] creates a list of graphics primitives and directives as a visual representation of the given element; layout is a list of rules with concrete values for all coordinates and style is an association with all style settings merged.

Optionally, for composite shapes that have the notion of connection points:

connector[e,conName] creates an anchor for the given element's connection point.