CL-DOT User Manual

Michael Weber

This manual is for CL-DOT, version 0.8.0-16-gbf364b.

Table of Contents

Introduction 1

1 Installation 2
1.1 External Dependencies 2

2 Usage 3
2.1 Node-Centric Graph Generation 3
2.2 Edge-Centric Graph Generation 5

3 Limitations 6

4 The CL-DOT Package 7
4.1 Variables 7
4.2 The `GRAPH-OBJECT` Protocol 7
4.3 Dot Attributes 8
4.3.1 Graph Attributes 8
4.3.2 Node Attributes 9
4.3.3 Edge Attributes 9
4.4 Generating Output 10
4.5 Classes 11
4.6 Deprecated Functionality 11
4.6.1 The `OBJECT` Protocol 11

5 Feedback and Support 12

6 Related Software 13

7 Copying 14

Indices 15
Variable Index 15
Function Index 15
Class Index 15

Introduction

CL-DOT is a small package for easily generating Dot (a program in the GraphViz suite) output from arbitrary Lisp data. It works with the following Common Lisp implementations:

- Allegro CL®
- GNU CLISP
- LispWorks
- SBCL

Original author is Juho Snellman, current maintainer is Michael Weber. The code is covered by the MIT license, see Chapter 7 [Copying], page 14.

1 Installation

CL-DOT together with this documentation can be downloaded from

`http://www.foldr.org/~michaelw/projects/cl-dot/`

It can be installed via QuickLisp:

```
(ql:quickload "cl-dot")
```

The source code is available via GitHub: `http://github.com/michaelw/cl-dot`

1.1 External Dependencies

CL-DOT has no dependencies on other systems, but it requires the GraphViz suite to be installed for rendering, and GNU Texinfo for preparing this documentation.

2 Usage

With CL-DOT, graphs can be constructed by starting from some *nodes* and recursively tracing edges until all reachable nodes and edges have been added.

2.1 Node-Centric Graph Generation

First, we define methods for the generic functions in the `GRAPH-OBJECT` protocol (see Section 4.2 [The GRAPH-OBJECT Protocol], page 7) for all objects that can appear in our graph. `graph-object-node` must be defined for all objects, the others have a default implementation. For example:

```
;; Conses
(defmethod cl-dot:graph-object-node ((graph (eql 'example)) (object cons))
  (make-instance 'cl-dot:node
                 :attributes '(:label "cell \\N"
                               :shape :box)))

(defmethod cl-dot:graph-object-points-to ((graph (eql 'example)) (object cons))
  (list (car object)
        (make-instance 'cl-dot:attributed
                       :object (cdr object)
                       :attributes '(:weight 3))))
;; Symbols
(defmethod cl-dot:graph-object-node ((graph (eql 'example)) (object symbol))
  (make-instance 'cl-dot:node
                 :attributes `(:label ,object
                               :shape :hexagon
                               :style :filled
                               :color :black
                               :fillcolor "#ccccff")))
```

Note that in this example, the first argument to all `GRAPH-OBJECT` functions, *graph*, is only used to segregate the rendering of cons cells and symbols from those of other uses of CL-DOT. However, it could also be used to look up node information which is stored external to *object*.

A call to `generate-graph-from-roots` generates an instance of `graph` for our data. From this graph instance, we can either generate dot-format output to some stream with `print-graph`, or call dot directly on the data with `dot-graph`. For example:

```
(let* ((data '(a b c #1=(b z) c d #1#))
       (dgraph (cl-dot:generate-graph-from-roots 'example (list data))))
  (cl-dot:dot-graph dgraph "test.png" :format :png))
```

We can also specify attributes for the whole graph:

```
(let* ((data '(a b c #1=(b z) c d #1#))
       (dgraph (cl-dot:generate-graph-from-roots 'example (list data)
                                                 '(:rankdir "LR"))))
  (cl-dot:dot-graph dgraph "test-lr.png" :format :png))
```

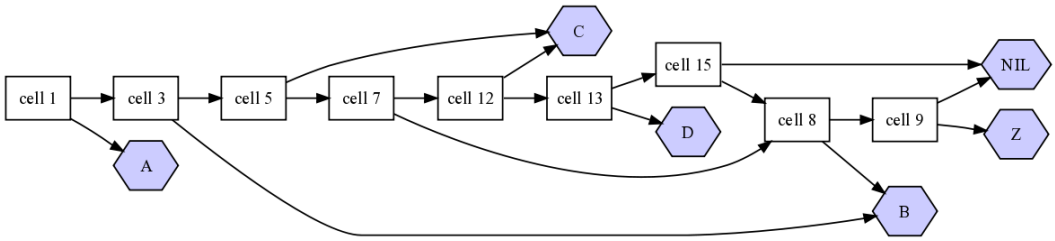

Figure 2.1: Graph of conses `(A B C #1=(B Z) C D #1#)`, generated by `dot-graph` with attributes `(:RANKDIR "LR")`

In order to render an undirected graph we can call `dot-graph` in the following way:

```
(cl-dot:dot-graph dgraph "test.png"
                  :format :png
                  :directed nil)
```

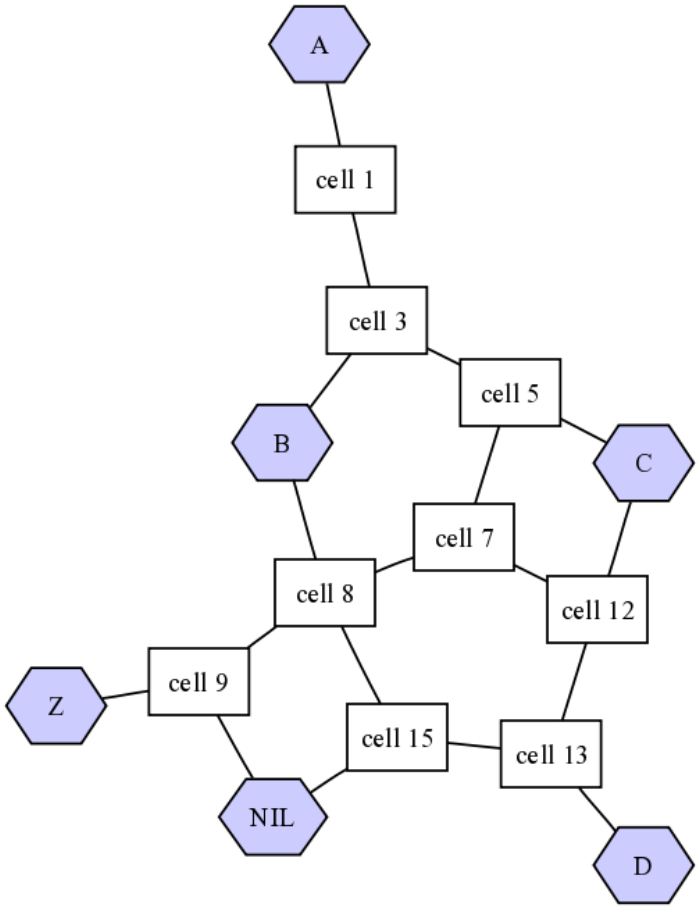

Figure 2.2: Graph of conses `(A B C #1=(B Z) C D #1#)`, generated by `dot-graph` with option `:directed nil`

When the *directed* keyword argument is set to `NIL` (the default value is `T`) `dot-graph` outputs an undirected graph instead of a directed. To do that it needs the `neato` program

from the Graphviz package, which is used to layout undirected graphs. The path to the `neato` program is stored in the `*neato-path*` special variable.

2.2 Edge-Centric Graph Generation

If a graph is stored as an edge list, the use of `graph-object-points-to` is not a good match. Instead, we can use `graph-object-edges` to return *edge specifications* (see [graph-object-edges], page 7) all edges which are part of the graph. For each object which appears as edge source or target, the appropriate functions of the `GRAPH-OBJECT` protocol are called.

```
;; Define how nodes are drawn
(defmethod cl-dot:graph-object-node ((graph (eql 'edge-example)) object)
  (make-instance 'cl-dot:node
                 :attributes (list :label (format nil "Node ~A" object)
                                   :shape :box
                                   :style :filled
                                   :color :black
                                   :fillcolor "#ccccff")))

;; Edges and their attributes
(defmethod cl-dot:graph-object-edges ((graph (eql 'edge-example)))
  #((a b (:color :red    :style :dashed))
    (b c (:color :blue   :style :dotted))
    (c a (:color :yellow :style :bold))))

(let ((dgraph (cl-dot:generate-graph-from-roots 'edge-example '()
                                                '(:rankdir "LR"))))
  (cl-dot:dot-graph dgraph "test-edges.png" :format :png))
```

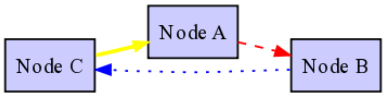

Figure 2.3: Graph generated via `graph-object-edges`

3 Limitations

Not all of the functionality of the GraphViz suite is accessible from CL-DOT. Patches which add more features are highly welcome.

4 The CL-DOT Package

4.1 Variables

`*dot-path*` [Special Variable]
`*neato-path*` [Special Variable]
Path to the `dot` and `neato` commands, both from the GraphViz suite.

4.2 The `GRAPH-OBJECT` Protocol

The `GRAPH-OBJECT` protocol is used to translate Lisp data into a graph representation suitable for rendering with the Dot program.

All protocol functions take as first parameter a *context object graph*. This allows to render objects like cons cells differently for different graphs.

Another use of the *graph* parameter is to look up node information needed for rendering. For example, if nodes are represented as integers and edges between nodes can be looked up in an external table, this table can be made accessible to the `GRAPH-OBJECT` functions via the *graph* parameter.

`generate-graph-from-roots` *graph objects* `&optional` *attributes* [Generic Function]
`generate-graph-from-roots T T &optional` *attributes* [Method]
Construct a *graph* with *attributes*, starting from *objects* (a sequence). Other functions of the `GRAPH-OBJECT` protocol are subsequently called on *objects* to discover other graph nodes and edges between them.

`graph-object-edges` *graph* [Generic Function]
`graph-object-edges T` [Method]
Returns a sequence of *edge specifications*.

An *edge specification* is a list `(from to [attributes])`, where *from* and *to* are objects of the graph and optional *attributes* is a *plist* of edge attributes, see Section 4.3.3 [Edge Attributes], page 9.

The default method returns an empty sequence.

`graph-object-knows-of` *graph object* [Generic Function]
`graph-object-knows-of T T` [Method]
Returns a sequence of objects that *object* knows should be part of the graph, but which it has no direct connections to.

The default method returns an empty sequence.

`graph-object-node` *graph object* [Generic Function]
Returns a `node` instance for *object*, or `NIL`. In the latter case the object will not be included in the constructed ouput, but it can still have an indirect effect via other protocol functions (e.g., `graph-object-knows-of`). This function will only be called once for each object during the generation of a graph.

`graph-object-pointed-to-by` *graph object* [Generic Function]
`graph-object-pointed-to-by T T` [Method]

Returns a sequence of objects to which the node of *object* should be connected. The edges will be directed from the other objects to this one.

To assign Dot attributes to the generated edges, each object can optionally be wrapped in an instance of class `attributed`.

The default method returns an empty sequence.

`graph-object-points-to` *graph object* [Generic Function]
`graph-object-points-to T T` [Method]

Returns a sequence of objects to which the `node` of *object* should be connected. The edges will be directed from *object* to the others.

To assign Dot attributes to the generated edges, each object can optionally be wrapped in an instance of class `attributed`.

The default method returns an empty sequence.

4.3 Dot Attributes

The rendering of Dot graphs, their nodes and edges can be influenced by *Dot attributes*. Attributes are represented as *keywords* of the same name. Multiple attributes can be given in form of a *plist*.

CL-DOT supports most Dot attributes, a detailed list follows. Attributes which are not recognized result in an *error* when generating a graph.

Most attributes have self-explanatory names, for more information we refer to the documentation of Dot.

4.3.1 Graph Attributes

Graph attributes can be given to `generate-graph-from-roots` and apply to the whole graph.

`:bgcolor` *text*
`:center` *integer*
`:color` *text*
`:edge` *edge-attribute*

The value of *edge-attribute* must be a single edge attribute, see Section 4.3.3 [Edge Attributes], page 9. For example:

```
(generate-graph-from-roots graph initial-states
                           '(:edge (:arrowhead :odot)))
```

`:layers` *text*
`:margin` *float*
`:mclimit` *float*
`:node` *node-attribute*

The value of *node-attribute* must be a single node attribute, see Section 4.3.2 [Node Attributes], page 9. For example:

```
(generate-graph-from-roots graph initial-states
                           '(:node (:shape :box)
                             :node (:color :red)))
```

`:nodesep` *float*
`:nslimit` *float*
`:ordering (:out)`
`:page` *text*
`:pagedir` *text*
`:rank (:same :min :max)`
`:rankdir ("LR" "RL" "BT")`
`:ranksep` *float*
`:ratio (:fill :compress :auto)`
`:rotate` *integer*
`:size` *text*

4.3.2 Node Attributes

`:color` *text*
`:fillcolor` *text*
`:fixed-size` *boolean*
`:fontname` *text*
`:fontsize` *integer*
`:height` *integer*
`:label` *label*

Provide a label for node. The string *label* may include escaped newlines '`\l`', '`\n`', or '`\r`' for left, center, and right justified lines. The string '`\N`' will be replaced by the node name.

Note that verbatim backslashes inside of Lisp strings must be escaped themselves. E.g., the attribute `:label "Node \\N\\l"` produces a left-justified node label which includes the node identifier.

By default, `#\Newline`-delimited lines contained in *label* are horizontally centered in Dot output. However, if the value of *label* is a list `(`*alignment text*`)`, then *alignment* specifies how lines in string *text* are rendered. *alignment* can be one of `:left`, `:center`, `:right`.

`:layer` *text*
`:shape` *shape*

shape can be one of the following keywords:

```
:record :plaintext :ellipse :circle :egg :triangle :box
:diamond :trapezium :parallelogram :house :hexagon :octagon
```

`:style (:filled :solid :dashed :dotted :bold :invis)`
`:width` *integer*

4.3.3 Edge Attributes

`:arrowhead` *arrow-spec*

arrow-spec can be one of the following keywords:

```
:none :normal :inv :dot :odot :invdot :invodot :tee :empty
:invempty :open :halfopen :diamond :odiamond :box :obox :crow
```

`:arrowtail` *arrow-spec*
: *arrow-spec* can be one of the following keywords:

```
:none :normal :inv :dot :odot :invdot :invodot :tee :empty
:invempty :open :halfopen :diamond :odiamond :box :obox :crow
```

`:color` *text*
`:constraint` *boolean*
`:decorate` *boolean*
`:dir (:forward :back :both :none)`
`:fontcolor` *text*
`:fontname` *text*
`:fontsize` *integer*
`:headclip` *boolean*
`:headlabel` *text*
`:label` *text*
: See [node-attr-label], page 9, for more information.

`:labeldistance` *integer*
`:labelfontcolor` *text*
`:labelfontname` *text*
`:labelfontsize` *integer*
`:layer` *text*
`:minlen` *integer*
`:port-label-distance` *integer*
`:samehead` *boolean*
`:sametail` *boolean*
`:style (:solid :dashed :dotted :bold :invis)`
`:tailclip` *boolean*
`:taillabel` *text*
`:weight` *integer*

4.4 Generating Output

`dot-graph` *graph outfile* `&key` *format directed* [Function]
: Renders *graph* (an instance of `graph`, see [graph], page 11) to *outfile*, by running the program in either `*dot-path*` or `*neato-path*`.

  When *directed* is `T` (the default) it will use the program specified in `*dot-path*` to render a directed graph. Otherwise, (when *directed* is `NIL`) `dot-graph` will render an undirected graph using the program specified in `*neato-path*`.

  The default *format* is Postscript.

`print-graph` *graph* `&optional` *stream* [Function]
: Prints a dot-format representation of *graph* (an instance of `graph`, see [graph], page 11) to *stream*.

4.5 Classes

`graph` [Standard Class]
A graph suitable for rendering with the Dot command. Instance of this class are most often generated with `generate-graph-from-roots` (see [generate-graph-from-roots], page 7) or `generate-graph` (see [generate-graph], page 11).

`node :attributes` *attributes* `:id` *id* [Standard Class]
A graph node with dot attributes (a *plist*, initarg *attributes*, see Section 4.3 [Dot Attributes], page 8) and an optional node identifier (initarg *id*, auto-generated by default).

`attributed :object` *object* `:attributes` *attributes* [Standard Class]
Wraps an object (initarg *object*) with edge attribute information (a *plist*, initarg *attributes*, see Section 4.3 [Dot Attributes], page 8). See Chapter 2 [Usage], page 3, for an example of how to use `attributed`.

4.6 Deprecated Functionality

The `OBJECT` protocol has been deprecated in favor of the more general `GRAPH-OBJECT` protocol (see Section 4.2 [The GRAPH-OBJECT Protocol], page 7), which allows objects to be presented differently for different graphs. For backwards compatibility, the `OBJECT` protocol functions are called by their respective `GRAPH-OBJECT` equivalents when `generate-graph` is used.

4.6.1 The `OBJECT` Protocol

`generate-graph` *object* `&optional` *attributes* [Generic Function]
`generate-graph T &optional` *attributes* [Method]
Construct a *graph* with *attributes* starting from *object*, using the `OBJECT` protocol.

The default method calls `generate-graph-from-roots` with a singleton list of *object*.

`object-knows-of` *object* [Generic Function]
`object-knows-of T` [Method]
The default method returns the empty list.

`object-node` *object* [Generic Function]
Returns a `node` instance for *object*, or `NIL`. In the latter case the object will not be included in the constructed ouput, but it can still have an indirect effect via other protocol functions (e.g., `object-knows-of`). This function will only be called once for each object during the generation of a graph.

`object-pointed-to-by` *object* [Generic Function]
`object-pointed-to-by T` [Method]
The default method returns the empty list.

`object-points-to` *object* [Generic Function]
`object-points-to T` [Method]
The default method returns the empty list.

5 Feedback and Support

Please direct bug reports, patches, questions, and any other feedback to Michael Weber. A small check list helps to stream-line the process of submitting patches:[1]

- When sending bug reports, please include a small test case.
- Please send patches in *unified* format. They can be created with `diff -u ...`, for example.
- Do not use `TAB` characters for indentation.
- Every new function you add should have a reasonable documentation string. If you change an existing function, change its docstring as well if necessary. The same applies to global variables, classes, class slots, and everything else you can attach a docstring to.
- If your patch is exporting new functionality or changing exported functionality, please update the library's documentation as well.
- If you modify existing behavior, always try to be backwards compatible or to at least provide a simple transition path for users of previous releases.

[1] lifted from `http://weitz.de/patches.html`

6 Related Software

The following Lisp projects provide similar functionality:

cl-graph This library has built-in support for the Dot output format. If we start out with CL-GRAPH data structures, this is probably the easiest way to render them with Dot.

cl-graphviz

This project provides a CFFI bindings to GraphViz. It provides richer access to (low-level) GraphViz functionality than CL-DOT. Also, it ties in with cl-graph.

Quoting the main developer, Attila Lendvai:[1]

> "[. . .] if you only want to layout a few graphs from the repl then trivial-shell and the utils in cl-graph are your friends to exec the dot binary. cl-graphviz only helps if you want to have a web service or something and want to avoid exec'ing."

s-dot This library allows to translate graphs specified in *s-expressions* to the Dot format, and also render them by calling the `dot` command.

[1] Message-Id: <49d30481-63ee-475e-be17-07cc684f2a56@w34g2000hsg.googlegroups.com>

7 Copying

This manual is for CL-DOT, version 0.8.0-16-gbf364b.

Indices

Variable Index

dot-path 7
neato-path 7

Function Index

D

dot-graph 10

G

generate-graph 11
generate-graph-from-roots 7
graph-object-edges 7
graph-object-knows-of 7
graph-object-node 7
graph-object-pointed-to-by 8
graph-object-points-to 8

O

object-knows-of 11
object-node 11
object-pointed-to-by 11
object-points-to 11

P

print-graph 10

Class Index

A

attributed 11

G

graph 11

N

node 11