Preface

This is a book about hacking in ocaml. It's assumed that you already understand the underlying theory. Happy hacking Most parts are filled with code blocks, I will add some comments in the future. Still a book in progress. Don't distribute it.

☺

Acknowledgements

write later

Contents

Todo list

Chapter 1

Tool Chain

Chapter 2

Lexing

Chapter 3

Parsing

Chapter 4

Camlp4

Camlp4 stands for Preprocess-Pretty-Printer for `OCaml`, it's extremely powerful and hard to grasp as well. It is a source-to-source level translation tool.

Chapter 5

Libraries

5.3 ocamlgraph

ocamlgraph is a sex library which deserve well-documentation.

1. simple usage in the module *Graph.Pack.Digraph*

```
se_str "label" "PDig.V";;
```

```
type label = int
val create : label -> t
val label : t -> label
```

Follow this file, you could know how to build a graph, A nice trick, to bind open command to use graphviz to open the file, then it will do the sync automatically

```
module PDig = Graph.Pack.Digraph
let g = PDig.Rand.graph ~v:10 ~e:20 ()
(* get dot output file *)
let _ = PDig.dot_output g "g.dot"
(* use gnu/gv to show *)
let show_g = PDig.display_with_gv;;

let g_closure = PDig.transitive_closure ~reflexive:true g
(** get a transitive closure *)
let _ = PDig.dot_output g_closure "g_closure.dot"

let g_mirror = PDig.mirror g
let _ = PDig.dot_output g_mirror "g_mirror.dot"

let g1 = PDig.create ()
let g2 = PDig.create ()


let [v1;v2;v3;v4;v5;v6;v7 ] = List.map PDig.V.create [1;2;3;4;5;6;7]

let _ = PDig.( begin
  add_edge g1 v1 v2;
  add_edge g1 v2 v1;
  add_edge g1 v1 v3;
  add_edge g1 v2 v3;
  add_edge g1 v5 v3;
  add_edge g1 v6 v6;
  add_vertex g1 v4
  end
)

let _ = PDig.( begin
  add_edge g2 v1 v2;
  add_edge g2 v2 v3;
  add_edge g2 v1 v4;
  add_edge g2 v3 v6;
  add_vertex g2 v7
end
)

let g_intersect = PDig.intersect g1 g2
let g_union = PDig.union g1 g2

let _ =
  PDig.(
    let f = dot_output in begin
    f g1 "g1.dot";
    f g2 "g2.dot";
    f g_intersect "g_intersect.dot";
    f g_union "g_union.dot"
    end
  )
```

and you can *#u "open *.dot"*, so nice

```
module PDig = Graph.Pack.Digraph
sub_modules "PDig";;
```

```
    module V :
    module E :
    module Mark :
    module Dfs :
    module Bfs :
    module Marking : sig val dfs : t -> unit val has_cycle : t -> bool end
    module Classic :
    module Rand :
    module Components :
    module PathCheck :
    module Topological :
```

Different modules have corresponding algorithms

2. hierachical

```
sub_modules "Graph" (** output too big *)
```

idea. can we draw a tree graph for this??

Graph.Pack requires its label being integer

```
sub_modules "Graph.Pack"
```

```
module Digraph :
    module V :
    module E :
    module Mark :
    module Dfs :
    module Bfs :
    module Marking :
    module Classic :
    module Rand :
    module Components :
    module PathCheck :
    module Topological :
module Graph :
    module V :
    module E :
    module Mark :
    module Dfs :
    module Bfs :
    module Marking :
    module Classic :
    module Rand :
    module Components :
    module PathCheck :
    module Topological :
```

3. hierachical for undirected graph

```
Graph.Pack.(Di)Graph
Undirected imperative graphs with edges and vertices labeled with integer.
Graph.Imperative.Matrix.(Di)Graph
Imperative Undirected Graphs implemented with adjacency matrices, of course integer(Matrix)


Graph.Imperative.(Di)Graph
Imperative Undirected Graphs.
Graph.Persistent.(Di)Graph
Persistent Undirected Graphs.
```

Here we have functor *Graph.Imperative.Graph.Concrete*, *Graph.Imperative.Graph.Abstract*, *Graph.Imperative.Graph.ConcreteLabeled*, *Graph.Imperative.Graph.AbstractLabeled*

we see that

```
module Abstract:
functor (V : Sig.ANY_TYPE) -> Sig.IM  with type V.label = V.t and type E.label
    = unit

module AbstractLabeled:
functor (V : Sig.ANY_TYPE) ->
functor (E : Sig.ORDERED_TYPE_DFT) -> Sig.IM  with type V.label = V.t and type
    E.label = E.t

module Concrete:
functor (V : Sig.COMPARABLE) -> Sig.I  with type V.t = V.t and type V.label = V
    .t and type E.t = V.t * V.t
    and type E.label = unit

module ConcreteBidirectional:
functor (V : Sig.COMPARABLE) -> Sig.I  with type V.t = V.t and type V.label = V
    .t and type E.t = V.t * V.t
and type E.label = unit

module ConcreteBidirectionalLabeled:
functor (V : Sig.COMPARABLE) ->
functor (E : Sig.ORDERED_TYPE_DFT) -> Sig.I  with type V.t = V.t and type V.
    label = V.t
and type E.t = V.t * E.t * V.t and type E.label = E.t

module ConcreteLabeled:
functor (V : Sig.COMPARABLE) ->
functor (E : Sig.ORDERED_TYPE_DFT) -> Sig.I  with type V.t = V.t and type V.
    label = V.t
      and type E.t = V.t * E.t * V.t and type E.label = E.t
```

so, as soon as you want to label your vertices with strings and your edges with floats, you should use functor. Take ConcreteLabeled as an example

```
module V = struct
  type t = string
  let compare = Pervasives.compare
  let hash = Hashtbl.hash
  let equal = (=)
end
module E = struct
  type t = float
  let compare = Pervasives.compare
  let default = 0.0
end
module X = Graph.Imperative.Graph.ConcreteLabeled (V) (E);;
module Y = Graph.Imperative.Digraph.ConcreteLabeled (V) (E);;

(**
    val add_edge : t -> vertex -> vertex -> unit
    val add_edge_e : t -> edge -> unit
    val remove_edge : t -> vertex -> vertex -> unit
    val remove_edge_e : t -> edge -> unit

    Not only that, but the V and E structure will work for
    persistent and directed graphs that are concretelabeled,
```

```
    and you can switch by replacing Imperative with Persistent
    , and Graph with Digraph.
    *)

module W = struct
  type label = float
  type t = float
  let weight x = x (* edge label -> weight *)
  let compare = Pervasives.compare
  let add = (+.)
  let zero = 0.0
  end

module Dijkstra = Graph.Path.Dijkstra (X) (W);;
```

4. another example (edge unlabeled, directed graph)

```
open Graph
module V = struct
  type t = string
  let compare = Pervasives.compare
  let hash = Hashtbl.hash
  let equal = (=)
end
module G = Imperative.Digraph.Concrete (V)
let g = G.create ()
let _ = G.( begin
  add_edge g "a" "b";
  add_edge g "a" "c";
  add_edge g "b" "d";
  add_edge g "b" "d"
end )
module Display = struct
  include G
  let vertex_name v = (V.label v)
  let graph_attributes _ = []
  let default_vertex_attributes _ = []
  let vertex_attributes _ = []
  let default_edge_attributes _ = []
  let edge_attributes _ = []
  let get_subgraph _ = None
end
module Dot_ = Graphviz.Dot(Display)
let _ =
  let out = open_out "g.dot" in
  finally (fun _ -> close_out out) (fun g ->
    let fmt =
      (out |> Format.formatter_of_output) in
    Dot_.fprint_graph fmt g ) g
```

It seems that Graphviz.Dot is used to display directed graph, Graphviz.Neato is used to display undirected graph.

here is a useful example to visualize the output generated by ocamldep.

```
open Batteries_uni
open Graph
module V = struct
  type t = string
  let compare = Perva
  let hash = Hashtbl.
  let equal = (=)
end
module StringDigraph
module Display = stru
  include StringDigra
  open StringDigraph
  let vertex_name v =
  let graph_attribute
  let default_vertex_
  let vertex_attribut
  let default_edge_at
  let edge_attributes
  let get_subgraph _
end

module DisplayG = Gra


let dot_output g file
  let out = open_out
  finally (fun _ -> c
    let fmt =
      (out |> Format.
    DisplayG.fprint_g


let g_of_edges edges
  let g = create () i
  let _ = Stream.iter
  g
)

let line = "path.ml:

let edges_of_line lin
  try
    let (a::b::res) =
      Pcre.split ~pat
    let v_a =
      let _ =  a.[0]<
      a  in
    let v_bs =
      (Pcre.split ~pa
    let edges = List.
    edges
  with exn -> invalid

let lines_stream_of_c
```

5.16 Modules

Chapter 6

Runtime

Chapter 7

GC

Should be re-written later

Chapter 8

Object-oriented

Write later

Chapter 9

Language Features

9.9 The module Language

write later

Chapter 10

subtle bugs

Chapter 11

Interoperating With C

Write later

Chapter 12

Pearls

Chapter 13

XX

Chapter 14

Topics