

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

Preface	3
Acknowledgements	5
1 Tool Chain	17
1.1 Ocamlbuild	18
multiple directories	21
grouping targets	22
With lex yacc, ocamlfind	23
1.1.1 Principles	24
1.1.2 Write plugin	25
Samples	28
Mixing with C stubs	30
1.1.3 OCamlbuild in the toplevel	36
1.1.4 Building interface files	37
Interaction with Git	37
1.2 Godi,otags	38
1.2.1 CheatSheet	38
1.3 Ocamlfind	39
1.3.1 CheatSheet	39
1.3.2 META file	39
1.4 toplevel	40
1.4.1 directives	40

1.4.2	Module toplevel	40
1.4.3	Env	41
1.5	OcamlDoc	43
1.6	ocamlmktop	46
1.7	Standard OCaml Develop Tools	47
1.8	ocamlmktop	49
1.9	Git	51
2	Lexing	53
2.1	Lexing	54
2.1.1	Ulex interface	59
2.2	Ocamllex	67
3	Parsing	71
3.1	Ocamlyacc	72
3.2	MENHIR Related	86
4	Camlp4	89
4.1	Predicate Parser	90
4.2	Basic Structure	91
4.2.1	Experimentation Environment	91
4.2.2	Camlp4 Modules	91
4.2.3	Camlp4 Make Toplevel	92
4.2.4	Command Options	95
4.2.5	Module Components	96
4.2.6	Simple Experiment	97
4.3	Camlp4 SourceCode Exploration	98
4.3.1	Camlp4 PreCast	98
4.3.2	OCamlInitSyntax	100
4.3.3	Camlp4.Sig	105
4.3.4	Camlp4.Struct.Camlp4Ast.mlast	105
4.3.5	AstFilters	106

4.3.6	Camlp4.Register	106
4.3.7	Camlp4Ast	111
4.3.8	TestFile	139
4.4	Extensible Parser	140
4.4.1	Examples	140
4.4.2	Mechanism	141
4.4.3	Parsing OCaml using Camlp4	146
	Fully Utilize Camlp4 Parser and Printers	146
	Otags Mini	146
	Parsing Json AST	148
4.5	STREAM PARSER	150
4.6	Grammar	151
4.6.1	LEVEL	153
4.6.2	Grammar Modification	153
	Example: Expr Parse Tree	156
4.7	QuasiQuotations	161
4.7.1	Quotation Introduction	162
4.7.2	Quotation Expander	163
	Lambda Example	167
4.8	Ast Transformation	171
4.9	Quotation Cont	174
4.9.1	Quotation module	174
4.10	Antiquotation Expander	176
4.11	Revised syntax	180
4.12	Filters in camlp4	186
4.12.1	Map Filter	186
4.12.2	Filter Examples	186
	Example: Map Filter	186
	Linking Problem	188
	Example: Add Zero	189
	Fold filter	189

Meta filter	189
Lift filter	191
Macro Filter	191
4.12.3 Example Tuple Map	192
4.12.4 Location Strip filter	192
4.12.5 Camlp4Profiler	193
4.12.6 Camlp4TrashRemover	193
4.12.7 Camlp4ExceptionTracer	193
4.13 Examples	194
4.13.1 Pa_python	194
4.13.2 Pa_list	199
4.13.3 Pa_abstract	201
4.13.4 Pa_apply	202
4.13.5 Pa_ctyp	202
4.13.6 Pa_exception_wrapper	203
4.13.7 Pa_exception_tracer	206
4.13.8 Pa_freevars	207
4.13.9 Pa_freevars_filter	207
4.13.10 Pa_global_handler	207
4.13.11 Pa_holes	207
4.13.12 Pa_minimm	207
4.13.13 Pa_plus	207
4.13.14 Pa_zero	207
4.13.15 Pa_printer	207
4.13.16 Parse_arith	208
4.13.17 Pa_estring	208
4.13.18 Pa_holes	217
4.14 Useful links	218
4.15 Camlp4 CheatSheet	219
4.15.1 Camlp4 Transform Syntax	219
Example: semi opaque	219

4.15.2	Parsing	220
5	Libraries	221
5.2	Format	223
5.2.1	Indentation Rules	224
5.2.2	Boxes	224
5.2.3	Directives	226
5.2.4	Example: Print	226
5.3	ocamlgraph	227
5.1	batteries	222
	syntax extension	222
5.1.1	Dev	222
5.1.2	BOLT	223
5.2	Mikmatch	224
5.3	pa-do	237
5.4	num	238
5.5	caml-inspect	239
5.7	pa-monad	253
5.8	bigarray	257
5.9	sexplib	258
5.10	bin-prot	261
5.11	fieldslib	262
5.12	variantslib	263
5.13	delimited continuations	264
5.14	shcaml	270
5.15	deriving	271
5.16	Modules	272
6	Runtime	275
6.1	ocamlrun	276
6.2	FFI	278
6.2.1	Data representation	279

6.2.2	Caveats	289
7	GC	291
8	Object-oriented	299
8.1	Simple Object Concepts	300
8.2	Modules vs Objects	304
8.3	More about class	305
9	Language Features	307
9.1	Stream Expression	308
9.2	GADT	312
9.3	First Class Module	313
9.4	Pahantom Types	317
9.4.1	Useful links	324
9.5	Positive types	325
9.6	Private Types	326
9.7	Subtyping	328
9.8	Explicit Nameing Of Type Variables	329
9.9	The module Language	330
10	subtle bugs	331
10.1	Reload duplicate modules	332
10.2	debug	334
10.3	Debug Cheat Sheet	335
11	Interoperating With C	337
12	Pearls	339
12.1	Write Printf-Like Function With Ksprintf	340
12.2	Optimization	340
12.3	Weak Hashtbl	340
12.4	Bitmatch	340

12.5 Interesting Notes	341
12.6 Polymorphic Variant	342
13 XX	347
13.0.1 tricks	348
13.0.2 ocaml blogs	352
14 Topics	353
14.1 First Order Unification	354
14.1.1 First-order terms	354
14.1.2 Substitution	355
14.1.3 Unification in Various areas	355
14.1.4 Occurs check	356
14.1.5 Unification Examples	356
14.1.6 Algorithm	356
14.2 LLVM	360

Todo list

write later	5
mlpack file	20
Glob Patterns	23
parser-help to coordinate menhir and ulex	66
predicate parsing stuff	90
Should be re-written later	291
Write later	306
read ml 2011 workshop paper	312
Read the slides by Jacques Garrigue	313
write later with subtyping	325
write later	330
polymorphic comparison	332
Write later	337

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*

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

```
1  type label = int
2  val create : label -> t
3  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

and you can `#u` “open `*.dot`”, so nice

```

1 module PDig = Graph.Pack.Digraph
2 let g = PDig.Rand.graph ~v:10 ~e:20 ()
3 (* get dot output file *)
4 let _ = PDig.dot_output g "g.dot"
5 (* use gnu/gv to show *)
6 let show_g = PDig.display_with_gv;;
7
8 let g_closure = PDig.transitive_closure ~reflexive:true g
9 (** get a transitive closure *)
10 let _ = PDig.dot_output g_closure "g_closure.dot"
11
12 let g_mirror = PDig.mirror g
13 let _ = PDig.dot_output g_mirror "g_mirror.dot"
14
15 let g1 = PDig.create ()
16 let g2 = PDig.create ()
17
18
19 let [v1;v2;v3;v4;v5;v6;v7] = List.map PDig.V.create [1;2;3;4;5;6;7]
20
21 let _ = PDig.( begin
22   add_edge g1 v1 v2;
23   add_edge g1 v2 v1;
24   add_edge g1 v1 v3;
25   add_edge g1 v2 v3;
26   add_edge g1 v5 v3;
27   add_edge g1 v6 v6;
28   add_vertex g1 v4
29 end
30 )
31
32 let _ = PDig.( begin
33   add_edge g2 v1 v2;
34   add_edge g2 v2 v3;
35   add_edge g2 v1 v4;
36   add_edge g2 v3 v6;
37   add_vertex g2 v7
38 end
39 )
40
41 let g_intersect = PDig.intersect g1 g2
42 let g_union = PDig.union g1 g2
43
44 let _ =
45   PDig.( begin
46     let f = dot_output in begin
47       f g1 "g1.dot";
48       f g2 "g2.dot";
49       f g_intersect "g_intersect.dot";
50       f g_union "g_union.dot"
51     end
52   )

```



```

1 module PDig = Graph.Pack.Digraph
2 sub_modules "PDig";;

```

```

1   module V :
2   module E :
3   module Mark :
4   module Dfs :
5   module Bfs :
6   module Marking : sig val dfs : t -> unit val has_cycle : t -> bool end
7   module Classic :
8   module Rand :
9   module Components :
10  module PathCheck :
11  module Topological :

```

Different modules have corresponding algorithms

2. hierachical

```

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

```

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

Graph.Pack requires its label being integer

```

1 sub_modules "Graph.Pack"

```

```

1  module Digraph :
2      module V :
3      module E :
4      module Mark :
5      module Dfs :
6      module Bfs :
7      module Marking :
8      module Classic :
9      module Rand :
10     module Components :
11     module PathCheck :
12     module Topological :
13 module Graph :
14     module V :
15     module E :
16     module Mark :
17     module Dfs :
18     module Bfs :
19     module Marking :
20     module Classic :
21     module Rand :
22     module Components :
23     module PathCheck :
24     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)

```

1 open Graph
2 module V = struct
3   type t = string
4   let compare = Pervasives.compare
5   let hash = Hashtbl.hash
6   let equal = (=)
7 end
8 module G = Imperative.Digraph.Concrete (V)
9 let g = G.create ()
10 let _ = G.( begin
11   add_edge g "a" "b";
12   add_edge g "a" "c";
13   add_edge g "b" "d";
14   add_edge g "b" "d"
15 end )
16 module Display = struct
17   include G
18   let vertex_name v = (V.label v)
19   let graph_attributes _ = []
20   let default_vertex_attributes _ = []
21   let vertex_attributes _ = []
22   let default_edge_attributes _ = []
23   let edge_attributes _ = []
24   let get_subgraph _ = None
25 end
26 module Dot_ = Graphviz.Dot(Display)
27 let _ =
28   let out = open_out "g.dot" in
29   finally (fun _ -> close_out out) (fun g ->
30     let fmt =
31       (out |> Format.formatter_of_output) in
32     Dot_.fprintf_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.

```

1  open Batteries_uni
2  open Graph
3  module V = struct
4      type t = string
5      let compare = Pervasives.compare
6      let hash = Hashtbl.hash
7      let equal = (=)
8  end
9  module StringDigraph
10 module Display = struct
11     include StringDigraph
12     open StringDigraph
13     let vertex_name v =
14     let graph_attribute
15     let default_vertex_
16     let vertex_attribut
17     let default_edge_at
18     let edge_attributes
19     let get_subgraph _
20 end
21
22 module DisplayG = Gra
23
24
25 let dot_output g file
26     let out = open_out
27     finally (fun _ -> c
28         let fmt =
29             (out |> Format.
30             DisplayG.fprint_g
31
32
33 let g_of_edges edges
34     let g = create () i
35     let _ = Stream.iter
36     g
37 )
38
39 let line = "path.ml:
40
41 let edges_of_line lin
42     try
43         let (a::b::res) =
44             Pcre.split ~pat
45         let v_a =
46             let _ = a.[0]<
47             a in
48         let v_bs =
49             (Pcre.split ~pa
50         let edges = List.
51         edges
52     with exn -> invalid
53
54 let lines_stream_of_c

```


5.16 Modules

Chapter 6

Runtime

Chapter 7

GC

Should
be re-
written
later

Chapter 8

Object-oriented

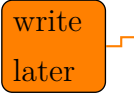
Write
later



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