# Contents

# 1 tool chain

## 1.1 ocamlbuild

### 1.1.1 directory hierachy

code : _build

- ob automatically creates a symbol link to the excutables it produces in the current directory
- ob copies the sources and compiles them in _build (default)
- hygiene rules at startup (.cmo, .cmi, or .o should appear outside of the _build) (-no-hygiene)
- ob must be invoked in the root directory

### 1.1.2 simple build

- ocamlbuild -quite xx.native – args
- ocamlbuild -quite -use-ocamlfind xx.native –args

**arguments**

- -log -verbose -clean
  _log file
- -cflags
  pass flags to ocamlc -cflags -I,+lablgtk,-rectypes
- -lflags
- -libs
  linking with external libraries *-libs unix,num* (you may need
  *-cflags -I,/usr/local/lib/ocaml*
  *-lflags -I,/usr/local/lib/ocaml*)
- -use-ocamlfind -pkgs oUnit
- working with lex yacc ocamlfind
  - lex yacc
    - .mll .mly supported by default
    - menhir (-use-menhir) or add true : `use_menhir`
- working with ocamlfind (syntax extension)

- in myocamlbuild.ml
  `<*.ml> : pkg_sexplib.syntax, pkg_batteries.syntax, syntax_camlp4o`
  here *sexpblib.syntax* and *syntax_camlp4o* is required by ocamlfind (.syntax) and will be translated to flag (-syntax camlp4o) for ocamlfind

### 1.1.3 _tags

- <**/*.ml>
  menas that .ml files in *current dir or sub dir*

- some default predicates
  <**/*.ml> <**/*.mli> <**/*.mlpack> <**/*.ml.depends> : ocaml
  <**/*.byte> : ocaml, byte, program

- or
  <*.ml> or <*.byte> or <*.native> : pkg_oUnit

- *ocamlbuild cares whitespace, take care when write _tags*

- regex

```
<**/*.{native,byte}> : use_unix
<{batMutex,batRMutex}.{ml,mli}>: threads

e1 or e2 , e1 and e2, not e, true ,false

e.g :
true:use_menhir
```

- create a file foo.itarget

```
bash$ cat foo.itarget
main.native
main.byte
stuff.docdir/index.html

--- ocamlbuild foo.otarget
```

- pack modules

```
create a file foo.mlpack in the directory foo
$ cat foo
Bar
Baz
```

- anoter typcial _tags file using syntax extension

```
<*.ml>: package(lwt.unix), package(lwt.syntax), syntax(camlp4o)
"prog.byte": package(lwt.unix)
-- only needs lwt.syntax when preprocessing
```

- document
  *when you use -keep-code; only document of exposed modules are kept, not of too use*

  ```
  flag ["ocaml"; "doc"] & S[A"-keep-code"];
  ```

  *try to learn ocamldoc config from batteries* idea: try to write a lightweight tool based on ocamldep

#### 1.1.4 godi

- `godi_console`
  - useful paths

    ```
    ./build/distfiles/godi-batteries
    ~/SourceCode/ML/godi/build/distfiles/ocaml-3.12.0/toplevel/
    ```

```
godi_make makesum
godi_make  install
godi_console info (godi_console list )
godi_add ~/SourceCode/ML/godi/build/packages/All/godi-calendar-2.03.tgz
godi_console perform -build godi-ocaml-graphics  >.log 2 >1
perform (fetch, extract, patch, configure, build, install)
```

### 1.2 ocamlfind

findlib

- utility
  - *ocamlfind browser -all*
  - *ocamlfind browser -package batteries*
- syntax

```
ocamlfind ocamldep -package camlp4,xstrp4 -syntax camlp4r file1.ml file2.ml

-- you see, ocamlfind can only handle flag camlp4r, flag camlp4o
-- so if you want to use other extensions,
-- use -package camlp4,xstrp4
-- i.e. -package camlp4.macro
```

- META

```
name="toplevel"
description = "toplevel hacking"
requires = ""
archive(byte) = "dir_top_level.cmo"
archive(native) = "dir_top_level.cmx"
version = "0.1"

all:
   @ocamlfind install toplevel META _build/*.cm[oxi]
clean:
   @ocamlfind remove toplevel
```

## 1.3 toplevel

### 1.3.1 directives

#directory "_build" ;; #directory "+camlp4" ;; #load "..."

- trace
- labels (ignore labels in function types)
- `warnings print_depth print_length`
- Toploop
  - re-direct

```
Toploop.execute_phrase (bool->formatter->Parsetree.toplevel_phrase->bool)
Toploop.read_interactive_input
{- : (string -> string -> int -> int * bool) ref =}
topdirs.cmi
```

  - `Hashtbl.keys Toploop.directive_table;;`
  - `Topdirs.dir_load`

```
Topdirs.dir_load
- : Format.formatter -> string -> unit = <fun>
dir_use : formatter -> string -> unit
dir_install_printer
dir_trace, dir_untrace, dir_untrance_all, load_file
dir_quit dir_cd
```

  - example in findlib

```
(* For Ocaml-3.03 and up, so you can do: #use "topfind" and get a
 * working findlib toploop.
 *)
(* First test whether findlib_top is already loaded. If not, load it now.
 * The test works by executing the toplevel phrase "Topfind.reset" and
 * checking whether this causes an error.
 *)
let exec_test s =
  let l = Lexing.from_string s in
  let ph = !Toploop.parse_toplevel_phrase l in
  let fmt = Format.make_formatter (fun _ _ _ -> ()) (fun _ -> ()) in
  try
    Toploop.execute_phrase false fmt ph
  with
      _ -> false
in

if not(exec_test "Topfind.reset;;") then (
  Topdirs.dir_load Format.err_formatter "/Users/bob/SourceCode/ML/godi/lib/ocam
  Topdirs.dir_load Format.err_formatter "/Users/bob/SourceCode/ML/godi/lib/ocam
);;
```

– topfind.ml ideas : we can write some utils to check code later

```
Hashtbl.add
    Toploop.directive_table
    "require"
    (Toploop.Directive_string
       (fun s ->
  protect load_deeply (Fl_split.in_words s)
       ))
;;


Hashtbl.add Toploop.directive_table "pwd" (Toploop.Directive_none (fun _ ->
  print_endline (Sys.getcwd ())));;
  - : unit = ()
# #pwd;;
/Users/bob/SourceCode/Notes
```

## 1.4 git

ignore set

- _log _build *.native *.byte *.d.native *.p.byte

## 1.5 parsing lexing

### 1.5.1 lexing

- use ulex (cauze unicode support), do't waste time in ocamllex (can handle CJK!!)
  - config

```
$ cat _tags
<*_ulex.ml> : syntax_camlp4o,pkg_ulex
<*_ulex.{byte,native}> : pkg_ulex

-- use default myocamlbuild.ml
-- ln -s ~/myocamlbuild.ml

-- make a symbol link pa_ulex.cma to camlp4 directory
-- this is actually not necessary
-- but sometimes for debugging purpose
-- as follows, this is pretty easy
camlp4o pa_ulex.cma  -printer OCaml test_ulex.ml -o test_ulex.ppo
```

  - syntax (does not support as syntax as ocamllex)

```
let regexp number = ['0'-'9'] +
-- let regexp line = [^ '\n']* ('\n' ?)
let u8l = Ulexing.utf8_lexeme

let rec lexer1 arg1 arg2 .. = lexer
|regexp -> action |..
and lexer2 arg1 arg2 .. = lexer
|regexp -> action |...
```

  - nice feature
    can roll back Ulexing.rollback lexbuf, so for string lexing, you can rollback one char, and plugin your string lexer
    but not generally usefull, ulex does not support shortest mode yet?
    so sometimes its semantics is wrong
  - use macro package
    since you need inline to do macro preprocessing so use syntax extension macro to inline your code, Attention! order matters

```
<*_ulex.ml> : syntax_camlp4o,pkg_ulex,pkg_camlp4.macro
<*_ulex.{byte,native}> : pkg_ulex
```

    Attention! **since you use ocamlbuild to build, then you need to copy you include fies to _build if you use relative path, otherwise you can use absolute path**

  - predefined regexp (copied from ocaml source code) lexer.ml

- Ulexing interface

  - roughly equivalent to the module Lexing, except that its lexbuffers handles Unicode code points(OCaml type:int in the range 0.. 0x10ffff) instead of bytes (OCamltype : char)
    *you can customize implementation for lex buffers*, define a module L which implements *start,next,mark, and backtrack and the Error exception.* They need not work on a type named lexbuf, you can use the type name you want. Then, just do in your *ulex-processed* source, before the first lexer specification`module Ulexing = L`
    Great! you can see that generated code *introducing Ulexing* very *late* and actually use very limited functions, other functions are provided for your convenience. and it did not have any type annotations, so you really can customize it. I think probably ocamllex can do the similar trick.

    ```
    val start : Ulexing.lexbuf -> unit
    val next : Ulexing.lexbuf -> int
    val mark : Ulexing.lexbuf -> int -> unit
    val backtrack : Ulexing.lexbuf -> int
    ```

  - interface

    ```
    type lexbuf
    exception Error
    exception InvalidCodepoint of int
    val create : (int array -> int -> int -> int ) -> lexbuf


    (* Unicode *)
    from_stream : int Stream.t -> lexbuf
    from_int_array : int array -> lexbuf

    (* 0..255 *)
    from_latin1_stream : char Stream.t -> Ulexing.lexbuf
    from_latin1_channel : Pervasives.in_channel -> Ulexing.lexbuf
    from_latin1_string : string -> Ulexing.lexbuf


    (*Utf8 encoded stream*)
    from_utf8_stream : char Stream.t -> Ulexing.lexbuf
    from_utf8_channel : Pervasives.in_channel -> Ulexing.lexbuf
    from_utf8_string : string -> Ulexing.lexbuf


    (** encoding is subject to change during lexing Note that bytes
    ```

```
have been consumed bye the lexer buffer are not re-interpreted
with the new encoding, in Ascii mode, non-Ascii bytes(ie >127) in the
stream raises an InvalidCodepoint exception
*)
from_var_enc_stream :
    Ulexing.enc Pervasives.ref -> char Stream.t -> Ulexing.lexbuf
from_var_enc_string :
    Ulexing.enc Pervasives.ref -> string -> Ulexing.lexbuf
from_var_enc_channel :
    Ulexing.enc Pervasives.ref -> Pervasives.in_channel -> Ulexing.lexbuf
type enc = Ulexing.enc = Ascii | Latin1 | Utf8
```

semantic action

```
lexeme_start : lexbuf -> int -- from 0
lexeme_end : lexbuf -> int
loc : lexbuf -> int * int -- (start,end)
lexeme_length : lexbuf -> int
lexeme : lexbuf -> int array
lexeme_char : lexbuf -> int -> int -- (may be more than 255)
sub_lexeme : lexbuf -> int -> int -> int array

latin1_lexeme : lexbuf -> string (*result encoded in Latin1*)
latin1_sub_lexeme
latin1_lexeme_char

utf8_lexeme
utf8_sub_lexeme

rollback : lexbuf -> unit
-- puts lexbuf back in its configuration before the last lexeme
-- was matched, it's then possible to plugin another lexer to parse
--

(** access to the internal buffer*)
get_buf : lexbuf -> int array
get_start : lexbuf -> int
get_pos : lexbuf -> int


-- internal
start,next,mark, backtrack
```

– inconvenience
did not handle line position, you have only global char position, but we are using

emacs, not matter too much

– hacking hand-coded some predefined regexps, copied and revised from ocaml compiler

```
let u8l = Ulexing.utf8_lexeme
let u8_string_of_int_array arr =
  Utf8.from_int_array arr 0 (Array.length arr)
let u8_string_of_int v =
  Utf8.from_int_array [|v|] 0 1

let report_error ?(msg="") lexbuf  =
  let (a,b) = Ulexing.loc lexbuf in
  failwith ((Printf.sprintf "unexpected error (%d,%d) : " a b )^ msg)

(** copied from ocaml 3.12.1 source code *)
let regexp newline = ('\010' | '\013' | "\013\010")
let regexp blank = [' ' '\009' '\012']
let regexp lowercase = ['a'-'z' '\223'-'\246' '\248'-'\255' '_']
let regexp uppercase = ['A'-'Z' '\192'-'\214' '\216'-'\222']

let regexp identchar =
  ['A'-'Z' 'a'-'z' '_' '\192'-'\214' '\216'-'\246' '\248'-'\255' '\'' '0'-'9']

let regexp symbolchar =
  ['!' '$' '%' '&' '*' '+' '-' '.' '/' ':' '<' '=' '>' '?' '@' '^' '|' '~']


let regexp decimal_literal =
  ['0'-'9'] ['0'-'9' '_']*
let regexp hex_literal =
  '0' ['x' 'X'] ['0'-'9' 'A'-'F' 'a'-'f']['0'-'9' 'A'-'F' 'a'-'f' '_']*
let regexp oct_literal =
  '0' ['o' 'O'] ['0'-'7'] ['0'-'7' '_']*
let regexp bin_literal =
  '0' ['b' 'B'] ['0'-'1'] ['0'-'1' '_']*
let regexp int_literal =
  decimal_literal | hex_literal | oct_literal | bin_literal
let regexp float_literal =
  ['0'-'9'] ['0'-'9' '_']*  ('.' ['0'-'9' '_']* )? (['e' 'E'] ['+' '-']? ['0'-'

let regexp blanks = blank +
let regexp whitespace = (blank | newline) ?
let regexp underscore = "_"
let regexp tilde = "~"
```

```
let regexp lident = lowercase identchar *

let regexp uidnet = uppercase identchar *

(** Handle string *)
let initial_string_buffer = Array.create 256 0
let string_buff = ref initial_string_buffer
let string_index = ref 0

let reset_string_buffer () =
  string_buff := initial_string_buffer;
  string_index := 0

(** store a char to the buffer *)
let store_string_char c =
  if !string_index >= Array.length (!string_buff) then begin
    let new_buff = Array.create (Array.length (!string_buff) * 2) 0 in
      Array.blit (!string_buff) 0 new_buff 0 (Array.length (!string_buff));
      string_buff := new_buff
  end;
  Array.unsafe_set (!string_buff) (!string_index) c;
  incr string_index

let get_stored_string () =
  let s = Array.sub (!string_buff) 0 (!string_index) in
  string_buff := initial_string_buffer;
  s


let char_for_backslash = function
  | 110 ->  10 (*'n' -> '\n'*)
  | 116 -> 9   (*'t' -> '\t' *)
  | 98  -> 8   (*'b' -> '\b'*)
  | 114 -> 13  (*'r' -> '\r' *)
  | c -> c
(** user should eat the first "\'"*)
let char_literal = lexer
  | newline "'" ->
    (Ulexing.lexeme_char lexbuf 0)
  |  [^ '\\' '\'' '\010' '\013'] "'" ->
    (* here may return a unicode we use *)
    (Ulexing.lexeme_char lexbuf 0)
  | "\\" ['\\' '\'' '"' 'n' 't' 'b' 'r' ' '] "'" ->
    (char_for_backslash (Ulexing.lexeme_char lexbuf 1 ))
  | "\\" ['0'-'9'] ['0'-'9'] ['0'-'9'] "'" ->
```

```
    let arr = Ulexing.sub_lexeme lexbuf 1 3 in
    (** Char.code '0' = 48 *)
    100*(arr.(0)-48)+10*(arr.(1)-48)+arr.(2)-48
  | "\\" 'x' ['0'-'9' 'a'-'f' 'A'-'F'] ['0'-'9' 'a'-'f' 'A'-'F'] "'" ->
    let arr = Ulexing.sub_lexeme lexbuf 2 2 in
    let v1 =
      if arr.(0) >= 97
      then (arr.(0)-87 ) * 16
      else if arr.(0) >= 65
      then (arr.(0)-55) * 16
      else (arr.(0) - 48) * 16 in
    let v2 =
      if arr.(1) >= 97
      then (arr.(1)-87 )
      else if arr.(1) >= 65
      then (arr.(1)-55)
      else (arr.(1) - 48) in
    (v1 + v2 )
  | "\\" _ ->
    let (a,b) = Ulexing.loc lexbuf in
    let l = Ulexing.sub_lexeme lexbuf 0 2  in
    failwith
    (Printf.sprintf
       "expecting a char literal (%d,%d) while %d%d appeared" a b l.(0) l.(1))
  | _ ->
    let (a,b) = Ulexing.loc lexbuf in
    let l = Ulexing.lexeme lexbuf in
    failwith
    (Printf.sprintf
       "expecting a char literal (%d,%d) while %d appeared" a b l.(0))




(** ocaml spuports multiple line string "a b \
    b" => interpreted as "a b b"
    actually we are always operation on an int
*)
let rec string = lexer
  |'"' -> () (* end *)

  | '\\' newline ([' ' '\t'] * ) ->
       string lexbuf

  | '\\' ['\\' '\'' '"' 'n' 't' 'b' 'r' ' '] ->
```

```
    store_string_char(char_for_backslash (Ulexing.lexeme_char lexbuf 1));
    string lexbuf
  | '\\' ['0'-'9'] ['0'-'9'] ['0'-'9'] ->
    let arr = Ulexing.sub_lexeme lexbuf 1 3 in
    let code = 100*(arr.(0)-48)+10*(arr.(1)-48)+arr.(2)-48 in
    store_string_char code ;
    string lexbuf
  | '\\' 'x' ['0'-'9' 'a'-'f' 'A'-'F'] ['0'-'9' 'a'-'f' 'A'-'F'] ->
    let arr = Ulexing.sub_lexeme lexbuf 2 2 in
    let v1 =
      if arr.(0) >= 97
      then (arr.(0)-87 ) * 16
      else if arr.(0) >= 65
      then (arr.(0)-55) * 16
      else (arr.(0) - 48) * 16 in
    let v2 =
      if arr.(1) >= 97
      then (arr.(1)-87 )
      else if arr.(1) >= 65
      then (arr.(1)-55)
      else (arr.(1) - 48) in
    let code = (v1 + v2 ) in
    store_string_char code ;
    string lexbuf
  | '\\' _ ->
    let (a,b) = Ulexing.loc lexbuf in
    let l = Ulexing.sub_lexeme lexbuf 0 2  in
    failwith
    (Printf.sprintf
       "expecting a string literal (%d,%d) while %d%d appeared" a b l.(0) l.(1)
    let (a,b) = Ulexing.loc lexbuf in
    let l = Ulexing.lexeme lexbuf in
    failwith
    (Printf.sprintf
       "expecting a string literal (%d,%d) while %d appeared" a b
        l.(0))
  | _ ->
    store_string_char (Ulexing.lexeme_char lexbuf 0);
    string lexbuf
(** you should provide '"' as entrance *)
let string_literal lexbuf =
  reset_string_buffer();
  string lexbuf;
  get_stored_string()
```

## 1.6 parsing

### 1.6.1 camlp4

- basics

```
bash-3.2$ camlp4 -where
/Users/bob/SourceCode/ML/godi/lib/ocaml/std-lib/camlp4
bash-3.2$ which camlp4
/Users/bob/SourceCode/ML/godi/bin/camlp4
```

```
find /Users/bob/SourceCode/ML/godi/bin -type f -perm -og+rx | grep camlp4
/Users/bob/SourceCode/ML/godi/bin/camlp4
/Users/bob/SourceCode/ML/godi/bin/camlp4boot
/Users/bob/SourceCode/ML/godi/bin/camlp4o
/Users/bob/SourceCode/ML/godi/bin/camlp4o.opt
/Users/bob/SourceCode/ML/godi/bin/camlp4of
/Users/bob/SourceCode/ML/godi/bin/camlp4of.opt
/Users/bob/SourceCode/ML/godi/bin/camlp4oof
/Users/bob/SourceCode/ML/godi/bin/camlp4oof.opt
/Users/bob/SourceCode/ML/godi/bin/camlp4orf
/Users/bob/SourceCode/ML/godi/bin/camlp4orf.opt
/Users/bob/SourceCode/ML/godi/bin/camlp4prof
/Users/bob/SourceCode/ML/godi/bin/camlp4r
/Users/bob/SourceCode/ML/godi/bin/camlp4r.opt
/Users/bob/SourceCode/ML/godi/bin/camlp4rf
/Users/bob/SourceCode/ML/godi/bin/camlp4rf.opt
/Users/bob/SourceCode/ML/godi/bin/mkcamlp4
/Users/bob/SourceCode/ML/godi/bin/safe_camlp4
```

so you have camlp4, camlp4o, camlp4of, camlp4oof, camlp4orf, camlp4r, camlp4rf tools to use some useful options for camlp4

```
camlp4 -h
-str parse string as an implementation
-unsafe generate unsafe access to array and strings
-QD Dump quotation expander result in case of syntax error
-o <file> output on <file> instead of standard output
-no_quot don't parse quotations, allowing to use, e.g, "<:>" as token
-loaded-modules
-parser <name> load the parser Camlp4Parsers/<name>.cm(o|a|xs)
-printer <name> load the printer Camlp4Printerss/<name>.cm(o|a|xs)
-- -printer o means print in original syntax
```

```
-filter <name> load the filter Camlp4Filters/<name>.cm(o|a|xs)


camlp4o -h
Options added by loaded object files
-add_locations Add locations as comment
-no_comments
-curry-constr (Use curried constructors)
-sep Use this string between parsers
```

- That reflextive is true means when extending the syntax of the host language will also extend the embeded one

| | host | embedded | reflective | 3.09 equivalent |
|---|---|---|---|---|
| camlp4of | original | original | Yes | N/A |
| camlp4rf | revised | revised | Yes | N/A |
| camlp4r-parser rq | revised | revised | No | camlp4r q_MLast.cmo |
| camlp4orf | original | revised | No | camlp4o q_MLast.cmo |
| camlp4oof | original | original | No | N/A |

loaded-modules

  - camlp4r
    - parser
      RP, RPP(RevisedParserParser)
    - printer
      OCaml
  - camlp4rf (extended from camlp4r)
    - parser
      RP,RPP, GrammarP, ListComprehension, MacroP, QuotationExpander
    - printer
      OCaml
  - camlp4o (extended from camlp4r)
    - parser
      OP, OPP, RP,RPP
  - camlp4of (extended from camlp4o)
    - parser
      GrammarParser, ListComprehension, MacroP, QuotatuinExpander
    - printer

- simple build (without ocamlbuild, ocamlfind)

```
ocamlc -pp camlp4o.opt error.ml
-- better error msg report
```

- simple example

```
camlp4of -str "let a = [x| x <- [1.. 10] ] "
let a = [ 1..10 ]

camlp4of -str "let q = <:str_item< let f x = x >>"
let q =
  Ast.StSem (_loc,
    (Ast.StVal (_loc, Ast.ReNil,
       (Ast.BiEq (_loc, (Ast.PaId (_loc, (Ast.IdLid (_loc, "f")))),
          (Ast.ExFun (_loc,
             (Ast.McArr (_loc, (Ast.PaId (_loc, (Ast.IdLid (_loc, "x")))),
                (Ast.ExNil _loc), (Ast.ExId (_loc, (Ast.IdLid (_loc, "x"))))))))))),
    (Ast.StNil _loc))
```

- hierachy
  - directory structure

```
|<.>
|--<Printers>
|--<Struct>
|----<Grammar>
```

  - Camlp4.PreCast

```
module Id = struct
  value name = "Camlp4.PreCast";
  value version = Sys.ocaml_version;
end;

type camlp4_token = Sig.camlp4_token ==
  [ KEYWORD       of string
  | SYMBOL        of string
  | LIDENT        of string
  | UIDENT        of string
  | ESCAPED_IDENT of string
  | INT           of int and string
  | INT32         of int32 and string
  | INT64         of int64 and string
  | NATIVEINT     of nativeint and string
  | FLOAT         of float and string
  | CHAR          of char and string
  | STRING        of string and string
  | LABEL         of string
  | OPTLABEL      of string
```

```
  | QUOTATION     of Sig.quotation
  | ANTIQUOT      of string and string
  | COMMENT       of string
  | BLANKS        of string
  | NEWLINE
  | LINE_DIRECTIVE of int and option string
  | EOI ];


(** Struct directory

has

module
  Loc, Dynloader

Functor
  Camlp4Ast.Make, Token.Make, Lexer.Make,
  Grammar.Static.Make, Quotation.Make

*)


module Loc = Struct.Loc;
module Ast = Struct.Camlp4Ast.Make Loc;
module Token = Struct.Token.Make Loc;
module Lexer = Struct.Lexer.Make Token;
module Gram = Struct.Grammar.Static.Make Lexer;
module DynLoader = Struct.DynLoader;
module Quotation = Struct.Quotation.Make Ast;
module MakeSyntax (U : sig end) = OCamlInitSyntax.Make Ast Gram Quotation;
module Syntax = MakeSyntax (struct end);
module AstFilters = Struct.AstFilters.Make Ast;
module MakeGram = Struct.Grammar.Static.Make;

module Printers = struct
  module OCaml = Printers.OCaml.Make Syntax;
  module OCamlr = Printers.OCamlr.Make Syntax;
  (* module OCamlrr = Printers.OCamlrr.Make Syntax; *)
  module DumpOCamlAst = Printers.DumpOCamlAst.Make Syntax;
  module DumpCamlp4Ast = Printers.DumpCamlp4Ast.Make Syntax;
  module Null = Printers.Null.Make Syntax;
end;
```

– file OcamlInitSyntax.ml

```
Make

(Ast:Sig.Camlp4Ast)

(Gram:
  Sig.Grammar.Static
with module Loc = Ast.Loc
with type Token.t = Sig.camlp4_token)

(Quotation : Sig.Quotation
with module Ast = Sig.Camlp4AstToAst Ast) :

Sig.Camlp4Syntax
with

module Loc = Ast.Loc
module Ast = Ast
module Gram = Gram
module Token = Gram.Token
module Quotation = Quotation
= struct


... bla bla

value a_LIDENT = Gram.Entry.mk "bla bla"



...
EXTEND_Gram
  top_phrase:
    [[ `EOI -> None ]]
  ;
END;

module AntiQuoteSyntax = Struct
  module LOC = Ast.Loc
  module Ast = Sig.Camlp4AstToAst Ast ; (** intersting *)
  (** Camlp4AstToAst the functor is a restriction
      functor. Takes a Camlp4Ast module and return it with some
      restrictions
   *)
  module Gram = Gram ;
```

```
  value antiquot_expr = Gram.Entry.mk "antiquot_expr";
  value antiquot_patt = Gram.Entry.mk "antiquot_patt";
  EXTEND_Gram
    antiquot_expr :
      [[ x = expr ; `EOI -> x ]] ;
    antiquot_patt :
      [[ x = patt ; `EOI -> x  ]]
  END;
  value parse_expr loc str = Gram.parse_string antiquot_expr loc str ;
  value parse_patt loc str = Gram.parse_string antiquot_patt loc str ;
end

value parse_implem ...
value parse_interf ...
value print_interf ...
value print_implem ...

module Quotation = Quotation ;
end
```

Notice Gram.Entry is **dynamic, extensible**

- experiment
  - toplevel

```
_loc : Loc.t

Camlp4.PreCast.Loc == Struct.Loc
Camlp4.PreCast.Ast == Struct.Camlp4Ast.Make Loc
Camlp4.PreCast.Token == Struct.Token.Make Loc
Camlp4.PreCast.Lexer == Struct.Lexer.Make Token
Camlp4.PreCast.Gram = Struct.Grammar.Static.Make Lexer
....

all Camlp4.PreCast stuff came from Struct.blablba

-- interesting bits
MakeSyntax (U:sig end) = OCamlInitSyntax.Make Ast Gram Qutation

let _loc = Loc.ghost
```

  - command line

```
ocaml
#camlp4r;
```

```
#load "camlp4rf.cma"

ocamlobjinfo `camlp4 -where`/camlp4fulllib.cma | grep -i unit
Unit name: Camlp4_import
Unit name: Camlp4_config
Unit name: Camlp4
Unit name: Camlp4AstLoader
Unit name: Camlp4DebugParser
Unit name: Camlp4GrammarParser
Unit name: Camlp4ListComprehension
Unit name: Camlp4MacroParser
Unit name: Camlp4OCamlParser
Unit name: Camlp4OCamlRevisedParser
Unit name: Camlp4QuotationCommon
Unit name: Camlp4OCamlOriginalQuotationExpander
Unit name: Camlp4OCamlRevisedParserParser
Unit name: Camlp4OCamlParserParser
Unit name: Camlp4OCamlRevisedQuotationExpander
Unit name: Camlp4QuotationExpander
Unit name: Camlp4AstDumper
Unit name: Camlp4AutoPrinter
Unit name: Camlp4NullDumper
Unit name: Camlp4OCamlAstDumper
Unit name: Camlp4OCamlPrinter
Unit name: Camlp4OCamlRevisedPrinter
Unit name: Camlp4AstLifter
Unit name: Camlp4ExceptionTracer
Unit name: Camlp4FoldGenerator
Unit name: Camlp4LocationStripper
Unit name: Camlp4MapGenerator
Unit name: Camlp4MetaGenerator
Unit name: Camlp4Profiler
Unit name: Camlp4TrashRemover
Unit name: Camlp4Top
```

  - Camlp4.Sig.ml

- revised syntax

```
'\'' -> '''

let x = 3              -> value  x = 42 ; (str_item) (do't forget ;)
let x = 3 in x + 8     -> let x = 3 in x + 7 (expr)

-- signature
val x : int  --> value x : int ;
```

```
-- abstract module types
module type MT --> module type MT = 'a

-- currying functor
type t = Set.Make(M).t   --> type t = (Set.Make M).t


--
e1;e2;e3    --> do{e1;e2;e3}

--
while e1 do e2 done --> while e1 do {e2;e3 }
for i = e1 to e2 do e1;e2 done --> for i = e1 to e2 do {e1;e2;e3}

--
() always needed

--
x::y -> [x::y]
x::y::z -> [x::[y::[z::t]]]
x::y::z::t -> [x;y;z::t]

--
match e with
[p1 -> e1
|p2 -> e2];

--
fun x -> x  --> fun [x->x]
--
value rec  fib = fun [
0|1  -> 1
|n -> fib (n-1) + fib (n-2)
];


fun x y (C z) -> t --> fun x y -> fun [C z -> t]
-- the curried pattern matching can be done with "fun", but
-- only irrefutable

-- legall
fun []
match e with []
try e with []
```

```
-- pattern after "let" and "value" must be irrefutable
let f (x::y) = ... --> let f = fun [ [x::y] ->  ... ]

--
x.f <- y  --> x.f := y
x:=!x + y --> x.val := x.val + y


--
int list --> list int

('a,bool) foo --> foo 'a bool

type 'a foo = 'a list list --> type foo 'a = list (list a)


--
int * bool --> (int * bool )


-- abstract type are represented by a unbound type variable

type 'a foo -->  type foo 'a = 'b

--
type t = A of i | B --> type t = [A of i | B]

-- empty is legal
type foo = []


type t= C of t1 * t2  --> type t = [C of t1 and t2]
C (x,y)               --> C x y

type t = D of (t1*t2)  --> type t = [D of (t1 * t2)]
D (x,y)                --> D (x,y)

--
type t = {mutable x : t1 } --> type t = {x : mutable t1}

--
(** the objects also have a revised syntax
    camlp4o pr_r.cmo file.ml
```

```
 *)


--
if a then b  --> if a then b else ()

--
a or b & c --> a || b && c


--
(+)  --> \+
(mod) --> \mod



-- new syntax
-- it's possible to group together several declarations
-- either in an interface or in an implementation by enclosing
-- them between "declare" and "end"
declare
  type foo = [Foo of int | Bar];
  value f : foo -> int ;
end ;

-- streams
[<'1;'2;s;'3>] -> [:`1; `2 ; s; `3 :]

parser [
 [: `Foo  :] -> e
|[: p = f :] -> f ]

--
parser []
match e with parser []


-- support where syntax
value e = c
  where c = 3 ;


-- parser
value x = parser [
 [: `1; `2  :] -> 1
```

```
|[: `1; `2 :] -> 2
];

-- object
class ['a,'b] point --> class point ['a,'b]
class c = [int] color --> class c = color [int]
-- signature
class c : int -> point --> class c : [int] -> point
--
method private virtual -->  method virtual private

--
object val x = 3 end --> object value x = 3; end

object constraint 'a = int end --> object type 'a = int ; end


-- label type

module type X = sig val x : num:int -> bool  end ;
-->
module type X = sig value x : ~num:int -> bool ; end;


--
~num:int
?num:int
```

- how it works

```
when your write code like this

open Camlp4.PreCast;
value _loc = Loc.ghost ;
value q  = <:str_item<value f x = x; >>;


(* two passes!!! *)
first camlp4rf pa_second_r.ml -printer o
get the output

open Camlp4.PreCast
let _loc = Loc.ghost
```

```
let q =
  Ast.StSem (_loc,
    (Ast.StVal (_loc, Ast.ReNil,
       (Ast.BiEq (_loc, (Ast.PaId (_loc, (Ast.IdLid (_loc, "f")))),
          (Ast.ExFun (_loc,
             (Ast.McArr (_loc, (Ast.PaId (_loc, (Ast.IdLid (_loc, "x")))),
                (Ast.ExNil _loc), (Ast.ExId (_loc, (Ast.IdLid (_loc, "x"))))))))))),
    (Ast.StNil _loc))

camlp4rf pa_second_r.ml -printer r


StSem -> Structure Semi

open Camlp4.PreCast;
value _loc = Loc.ghost;
value q =
  Ast.StSem _loc
    (Ast.StVal _loc Ast.ReNil
       (Ast.BiEq _loc (Ast.PaId _loc (Ast.IdLid _loc "f"))
          (Ast.ExFun _loc
             (Ast.McArr _loc (Ast.PaId _loc (Ast.IdLid _loc "x"))
                (Ast.ExNil _loc) (Ast.ExId _loc (Ast.IdLid _loc "x"))))))
    (Ast.StNil _loc);



so, you then ocamlc, and links (here your link extension
still needs camlp4.lib)
```

- some damn useful functions

```
Printers.OCaml.print_implem
Printers.OCamlr.print_implem

Camlp4.Sig.Printer :
  functor (Ast: Camlp4.Sig.Ast ) ->
    sig
      val print_interf :
        ?input_file: string -> ?output_file:string -> Ast.sig_item -> unit
      val print_implem :
        ?input_file: string -> ?output_file:string -> Ast.str_item -> unit
  end

Ast.tyOr_of_list : ctyp list -> ctyp
```

```
list_of_ctyp : ctyp -> ctyp list -> ctyp list
ctyp
  TySum of loc * ctyp

match_case =
  McNil of loc
  McOr of loc * match_case * match_case
  McArr  of loc * patt * expr * expr
  McAnt of loc * string
```

- quotations

```
[‘QUOTATION x -> Quotation.expand _loc x Quotation.DynAst.expr_tag
 ]
--
-- when it parsing to <tag< >>, will transfer control


add_quotation "sig_item" sig_item_quot ME.meta_sig_item MP.meta_sig_item

-- installs a quotation expander for the sig_item tag
-- the expander parses the quotation starting at the sig_item_quot
-- nonterminal in the parser, then runs the result through the antiquotation

-- inside a pattern
-- inside a expr
```

- so let's do something to dig
  - camlp4Ast mainly in the file Camlp4Ast.mlast

    Camlp4.Make (Loc) -> Sig.Camlp4Ast in the Make functor we have

    ```
    module Ast = struct
      include Sig.MakeCamlp4Ast Loc ;
    end;
    ```

    this inlcude the definition of Camlp4Ast.partial.ml most interesting ype definition. another intresting bit in this file is that

    ```
    class map = Camlp4MapGenerator.generated ;
    class fold = Camlp4FoldGenerator.generated ;
    ```

    The filter *Camlp4MapGenerator* reads *OCaml* type definitions and generate a class that implements a map traversal. The generated class have a method per type you can override to implement a *map traversal*. for example

```
open Camlp4.PreCast
let simplify = object
  inherit Ast.map as super
  method expr e = match super#expr e with
    |<:expr< $x$ + 0 >> | <:expr<0 + $x$ >> -> x
    | x -> x
end
in AstFileters.register_str_item_filter simplify#str_item
```

camlp4 provides other utilities to make life easier

```
open Camlp4.PreCast
let simplify = Ast.map_expr begin function
    |<:expr< $x$ + 0 >> | <:expr<0 + $x$ >> -> x
    | x -> x
end in AstFilters.register_str_item_filter simplify#str_item
(**
register_str_item_filter
register_sig_item_filter
register_topphrase_filter
*)
```

it's still not powerful, since you can only generate map travseral for OCaml types, you can do it!! *put your type definition just before* your macro

```
type t1 = ...
and t2 = ...
and tn = ...
;
class map = Camlp4MapGenerator.generated;
```

a simple example

```
cat _tags

<*_ulex.ml> : syntax_camlp4o,pkg_ulex,pkg_camlp4.macro
<*_ulex.{byte,native}> : pkg_ulex
<*_r.ml> : syntax_camlp4r, pkg_camlp4.quotations.r, pkg_camlp4.macro
"map_filter_r.ml" : pp(camlp4r -filter map)
<pa*.{cmo,byte,native}> : pkg_camlp4.lib
<*_o.ml> : syntax_camlp4o,pkg_sexplib.syntax

cat map_filter_r.ml
type a = [A of b | C ]
and b = [B of a | D ];
class map = Camlp4MapGenerator.generated;
(* output
type a = [ A of b | C ] and b = [ B of a | D ];
```

```
class map =
  object ((o : 'self_type))
    method b : b -> b = fun [ B _x -> let _x = o#a _x in B _x | D -> D ];
    method a : a -> a = fun [ A _x -> let _x = o#b _x in A _x | C -> C ];
    method unknown : ! 'a. 'a -> 'a = fun x -> x;
  end;
*)
```

actually in file Camlp4QuotationCommon.ml we also use this trick to handle `antiquot_expander`

```
value antiquot_expander = object
  inherit Ast.map as supre ;
  method patt : patt -> patt ...
  method expr : expr -> expr ...
```

so, who is in charge of invoking `antiquot_expander`, the function `add_quotation` in the definition of `add_quotation`, we have

```
let expand_expr loc loc_name_opt s =
  let ast = parse_quot_string entry_eoi loc s in
  let () = MetaLoc.loc_name.val := loc_name_opt in
  let meta_ast = mexpr loc ast in
  let exp_ast = antiquot_expander#expr meta_ast in
  exp_ast in
```

so, it first `parse_quot_string`, then do some transformation, that's how quotation works!!!, it will change to your customized quotation parser, and when it goes to antiquote syntax, it will go back to host langauge parser. Since the host language parser also support quotation syntax (**reflextive**), so you can nest your quotation whatever you want.

- benefits free ocamllexer, ocamlparser (parsed to camlp4ast actually), an easy to use yacc alternative, (extensible parser). and SYB generics (filters). It's a shame that ocaml does not support reflection very well?

- example

```
cat /usr/local/bin/oco
ledit -x -h ~/.ocaml_history ocaml dynlink.cma camlp4of.cma -warn-error +a-4-6-27..
# so we can do oco pretty easy
# suggestion: do it in terminal

open Camlp4.PreCast ;;
Gram.Entry.print Format.std_formatter Syntax.implem ;;
-- amazing, will print it

Gram.Entry.print Format.std_formatter Syntax.implem;;
```

```
implem: [ LEFTA
  [ "#"; a_LIDENT; opt_expr; semi
  | EOI
  | str_item; semi; SELF ] ]

Gram.Entry.print Format.std_formatter Syntax.top_phrase;;
top_phrase: [ LEFTA
  [ "#"; a_LIDENT; opt_expr; ";;"
  | EOI
  | LIST1 str_item; ";;" ] ]

Gram.Entry.print Format.std_formatter Syntax.phrase;;
phrase: [ LEFTA
  [ "#"; a_LIDENT; opt_expr; semi
  | str_item; semi ] ]

Gram.Entry.print Format.std_formatter Syntax.semi;;
semi: [ LEFTA
  [ ";;"
  |  ] ]

(** you can even print Syntax.expr, Syntax.str_item*)
```

you define your own ast, one way is to use MetaExpr to map your ast to Ocaml ast automatically, which is not interesting actually. (mainly used in QuasiQuoations), for this way you don't need to care the locations, meta_xx, will help you handle it . the parser get the location, and pass it to meta_xxx.

```
module Python_ast = struct
  type expr  =
    [Var of string
    |String of string  ]
  and stm   =
    [ Def of string and expr
    | Print of list expr ] ;
end;

include Python_ast;

open Camlp4.PreCast ;

module MetaExpr = struct
  include Camlp4Filters.MetaGeneratorExpr(Python_ast);
end ;
module MetaPatt = struct
```

```
  include Camlp4Filters.MetaGeneratorPatt(Python_ast);
end ;

open Camlp4.PreCast ;

value expr = Gram.Entry.mk "expr" ;
value stm = Gram.Entry.mk "stm" ;

module X = Gram ;

EXTEND X
  expr : [
    [v = LIDENT -> Var v
    |s = STRING ->  String s]
  ]
  ;
  stm: [
    [ "def"; v = LIDENT; "="; e=expr -> Def v e
    | "print"; es = LIST1 expr SEP "," -> Print es]
  ]
  ;
END;

(* value _ = Printf.printf "fuck";  *)

Gram.Entry.clear Syntax.expr;

EXTEND Gram
  Syntax.expr :
  [[ s = stm -> MetaExpr.meta_stm  _loc  s]];
END;
```

another way is to define your parser, your own ast, and your own mapping from your ast to OCaml ast. you still need not care the location, handle it to the mapping function

```
module Python_ast = struct
  type expr  =
    [Var of string
    |String of string  ]
  and stm   =
    [ Def of string and expr
    | Print of  list expr ] ;
end;
```

```
include Python_ast;
open Camlp4.PreCast ;
value meta_expr _loc = fun
  [Var str -> <:expr< $lid:str$ >>
  |String str -> <:expr< $str:str$ >>];
value concat_exprs _loc = fun
  [ [] -> failwith "concat_exprs"
  | [e::es] ->
    List.fold_left (fun e e' -> <:expr< $e$ ^ " " ^ $e'$  >>) e es
  ];
value meta_stm _loc = fun
  [Def str expr -> <:str_item< value $lid:str$ = $meta_expr _loc expr$ ; >>
  |Print es ->
    let es = List.map (fun e -> meta_expr _loc e) es in
    <:str_item< print_endline $concat_exprs _loc es $ >>
  ];
value expr = Gram.Entry.mk "expr" ;
value stm = Gram.Entry.mk "stm" ;
EXTEND Gram
  expr : [
    [v = LIDENT -> Var v
    |s = STRING ->  String s]
  ]
  ;
  stm: [
    [ "def"; v = LIDENT; "="; e=expr -> Def v e
    | "print"; es = LIST1 expr SEP "," -> Print es]
  ]
  ;
END;
(* value _ = Printf.printf "fuck";  *)
Gram.Entry.clear Syntax.str_item;
EXTEND Gram
  Syntax.str_item :
  [[ s = stm -> meta_stm  _loc  s]];
END;

$cast test_wiki2_r.ml
def name = "world" ;
print "hello", name ;

$camlp4rf -parser _build/wiki2_r.cmo test_wiki2_r.ml -printer r
(*
value name = "world";
print_endline ("hello" ^ (" " ^ name));
```

```
*)
```

another formal way to use the syntax extension is to use functor to register it , the whole file is a functor, we programed based on Camlp4.Sig

```
module Python_ast = struct
  type expr  =
    [Var of string
    |String of string  ]
  and stm   =
    [ Def of string and expr
    | Print of  list expr ] ;
end;
include Python_ast;
(* open Camlp4.PreCast ;  *)
module Id = struct
 value name = "python";
 value version = "0.1";
end;
module Minimal (Syntax : Camlp4.Sig.Camlp4Syntax ) = struct
  open Camlp4.Sig ;
  open Syntax ;
value meta_expr _loc = fun
  [Var str -> <:expr< $lid:str$ >>
  |String str -> <:expr< $str:str$ >>];
value concat_exprs _loc = fun
  [ [] -> failwith "concat_exprs"
  | [e::es] ->
    List.fold_left (fun e e' -> <:expr< $e$ ^ " " ^ $e'$  >>) e es
  ];
value meta_stm _loc = fun
  [Def str expr -> <:str_item< value $lid:str$ = $meta_expr _loc expr$ ; >>
  |Print es ->
    let es = List.map (fun e -> meta_expr _loc e) es in
    <:str_item< print_endline $concat_exprs _loc es $ >>
  ];
value expr = Gram.Entry.mk "expr" ;
value stm = Gram.Entry.mk "stm" ;
EXTEND Gram
  expr : [
    [v = LIDENT -> Var v
    |s = STRING ->  String s]
  ]
  ;
```

```
  stm: [
    [ "def"; v = LIDENT; "="; e=expr -> Def v e
    | "print"; es = LIST1 expr SEP "," -> Print es]
  ]
  ;
END;
(* value _ = Printf.printf "fuck";  *)
Gram.Entry.clear Syntax.str_item;
EXTEND Gram
  Syntax.str_item :
  [[ s = stm -> meta_stm  _loc  s]];
END;
  include Syntax ; (* Syntax -> Syntax we only care the side effect *)
end ;
value _ =
  let module M = Camlp4.Register.OCamlSyntaxExtension Id Minimal in ();
```

syntax extension works with both camlp4o and camlp4f, actually it's buggy.

```
ocaml dynlink.cma camlp4o.cma wiki2_r.cmo
        Objective Caml version 3.12.1

Camlp4 Parsing version 3.12.1

File "/Users/bob/.ocamlinit", line 1, characters 0-3:
Error: Parse error: illegal begin of use_file
# def name = "world";;
val name : string = "world"
#

-- error because we changed the syntax, it can not parse .ocamlinit any more

-- notice our mapping maps our ast into ocaml ast

-- previous our mapping maps our ast into ocaml ast (which represents our ast), so
-- needs your .mli file
-- we could build .mli by ocamlbuild xx.inferred.mli and then copy paste

-- wiki2_r.mli : use_camlp4_full (we are not using extension, just use the library)

  Gram.Entry.print - : Format.formatter -> 'a Camlp4.PreCast.Gram.Entry.t -> unit =
  Gram.Entry.clear
```

- Make a new grammar using the same lexer and token type

```
open Camlp4.PreCast ;;
module Gram = MakeGram(Lexer );;
```

- abbrev

```
camlp4 -parser r -parser rp -printer a
--  revised, revisedparserparser
-- == camlp4r

camlp4 -parser o -parser op -printer a
-- revisedparser, revisedparserparser, parser, parserparser
-- == camlp4o

-- the list of abbrevations for the parsers is in the file Camlp4Bin.ml

-- -printer a
-- either Camlp4OCamlPrinter (tty) or the Camlp4OCamlAstDumper (-printer p)


Gram.parse_string
-: 'a Gram.Entry.t -> Gram.Loc.t -> string -> 'a
```

- just use the parser

```
open Camlp4.PreCast;
value expression = Gram.Entry.mk "expression" ;
EXTEND Gram
  GLOBAL: expression ;
  expression : [
     "add" LEFTA
   [ x = SELF ; "+" ; y = SELF ->  x +  y
   | x = SELF ; "-" ; y = SELF ->  x -  y]
  | "mult" LEFTA
   [ x = SELF ; "*" ; y = SELF -> x * y
   | x = SELF ; "/" ; y = SELF -> x / y]
  | "pow" RIGHTA
   [ x = SELF ; "**" ; y = SELF -> int_of_float (float x ** float y) ]
  | "simple" NONA
   [ x = INT -> int_of_string x
   | "(" ; x = SELF ; ")" -> x ]
  ] ;
END;
value _ = Printf.printf "%d" (
  Gram.parse_string
```

```
    expression
    (Loc.mk "<string>" ) "3 +  ((4 - 2) + 28 * 3 ** 2) + (4 / 2)" )

;
(* (read_line ()) ;  *)

$cat _tags
<pa_*r.{ml,cmo,byte}> : pkg_dynlink , camlp4rf, use_camlp4_full
```

- keywords

```
EXTEND END

--  list of symbols, possible empty
LIST0 : LIST0 rule | LIST0 [ <rule definition> -> <action> ]

-- with a separator
LIST0 : LIST0 rule SEP <symbol>
| LIST0 [<rule definition > -> <action>] SEP <symbol>


  LIST1 rule
| LIST1 [<rule definition > -> <action > ]
| LIST1 rule SEP <symbol>
| LIST1 [<rule definition > -> <action >] SEP <symbol>


OPT <symbol>
SELF
TRY -- read the source code

FIRST LAST LEVEL level, AFTER level, BEFORE level
```

- links

```
http://brion.inria.fr/gallium/index.php/Abstract_Syntax_Tree
```

- define your ast, and map it to camlp4 ast

```
$cat vector_r.ml
open Sexplib ;
open Sexplib.Std ;  (* for some methods *)
```

```
type vec =
  [Scalar of string
  |Vector of  list string
  |Sum of vec and vec
  |ScalarProduct of vec and vec
  |Antiquot of string ]
with sexp ;
value (|>) x f = f x ;
value vec_to_string  vec =
    vec |> sexp_of_vec |> Sexp.to_string ;
$cat pa_vector_r.ml
open Camlp4.PreCast ;
open Vector_r ;
value rec meta_vec _loc = fun
  [Scalar s -> <:expr< $flo:s$ >>
  |Vector ls -> List.fold_right
    (fun x l -> <:expr< [ $flo:x$ :: $l$ ] >>)
    ls <:expr< [] >>
  |Sum l r -> <:expr< List.map2 (+.)  $meta_vec _loc l$   $meta_vec _loc r$ >>
  |ScalarProduct l0 r0 ->
    let l = meta_vec _loc l0
    and r = meta_vec _loc r0
    in match (l0,r0) with
    [(Scalar _ , Scalar _ ) ->  <:expr< $l$ *. $r$ >>
    |(Scalar _,  _) ->
      <:expr< List.map (fun x -> $l$ *. x) $r$ >>
    |(_, Scalar _) ->
      <:expr< List.map (fun x -> $r$ *. x) $l$>>
    | _ ->
      <:expr< List.fold_left (+.) 0. (List.map2 ( *. ) $l$ $r$ )>>
    ]

   |Antiquot s ->
     <:expr< $lid:s$ >> (** interesting *)
  ];

value expression = Gram.Entry.mk "expression";
EXTEND Gram
  GLOBAL: expression ;
  expression :
    [ "sum" LEFTA
      [x = SELF; "+"; y = SELF -> Sum x y ]
    | "scalar" LEFTA
      [x = SELF; "*"; y = SELF -> ScalarProduct x y ]
    | "simple" NONA
```

```
      ["("; e = SELF; ")" -> e
      | s = scalar -> Scalar s
      | v = vector -> v
      | s= LIDENT -> Antiquot s ]
    ];
  scalar :
    [[`INT (i,_) -> string_of_float (float i) (* for full information *)
        |`FLOAT (_,f) -> f ]
    ];
  vector :
    [[ "["; strs = LIST0 scalar SEP "," ; "]" -> Vector strs ]];
END;
Gram.Entry.clear Syntax.expr ; (* in the module Syntax *)
EXTEND Gram
  GLOBAL: Syntax.expr ;
  Syntax.expr:
    [[x = expression -> meta_vec _loc x ]];
END;
(** test parser *)
value _ =
    let _loc  = Loc.mk "<string>" in
    "[1,2,3]"
     |>  Gram.parse_string expression _loc
     |> vec_to_string  |> print_string ;
     (* |> sexp_of_vec |> Sexp.to_string |> print_string ;  *)

     (* |> vec_print |> print_string ; *)
     (* |>  sexp_of_vec |> string_of_sexp |> print_string ;  *)

(* [1,2,3] + [3,4,5];; *)
(* - : float list = [4.; 6.; 8.] *)


Register.loaded_modules;;

Camlp4.PreCast.Printers.OCaml.print_implem ;;

let module M = Camlp4.Printers.OCaml.Make Syntax in M.print_implem ;;
```

# 2 libraries

## 2.1 batteries

**syntax extension** Not of too much use , **Never use it in the toplevel**

- comprehension (M.filter, concat, map, filter_map, enum, of_enum)
  since it's at preproccessed stage, you can use some trick
  `let module Enum = List in` will change the semantics
  `let open Enum in` doesn't make sense, since it uses qualified name inside

#### 2.1.1 Dev

- make changes in both .ml and .mli files

## 2.2 Mikmatch

- benefit

```
% directly supported in toplevel
ocaml
#camlp4o ;;
% #load "pa_mikmatch_pcre.cma" ;; because I have a link
#require "mikmatch_pcre" ;;

camlp4of  -parser pa_mikmatch_pcre.cma test_mikmatch.ml -printer o
```

## 2.3 objsize

## 2.4 pa-do

- delimited overloading

## 2.5 Modules

- BatEnum
  - utilities

    ```
    range ~until:20 3
    filter, concat, map, filter_map
    (--), (--^) (|>) (@/) (/@)
    No_more_elements (*interface for dev to raise (in Enum.make next)*)
    icons, lcons, cons
    ```

  - don't play effects with enum
  - idea??? how about divide enum to two; one is just for iterator the other is for lazy evaluation. (iterator is lazy???)
- Set (*one comparsion, one container*)

```
Set.IntSet
Set.CharSet
Set.RopeSet
Set.NumStringSet

-- functions
split
union
-- why polymorphic set is dangerous?
-- because in Haskell, Eq a => is implicitly
-- you want to make your comparison method is unique
-- otherwise you union two sets, how to make sure they
-- they use the same comparison, here we use abstraction
-- types, one comparison, one container
```

we can not override polymorphic = behavior, polymorphic = is pretty bad practice for complex data structure, mostly not you want, so write compare by yourself

```
# Set.IntSet.(compare (of_enum (1--5))  (of_enum (List.enum [5;3;4;2;1])));;
- : int = 0
# Set.IntSet.(of_enum (1--5) = of_enum (List.enum [5;3;4;2;1]));;
- : bool = false
```

- caveat
  - module syntax

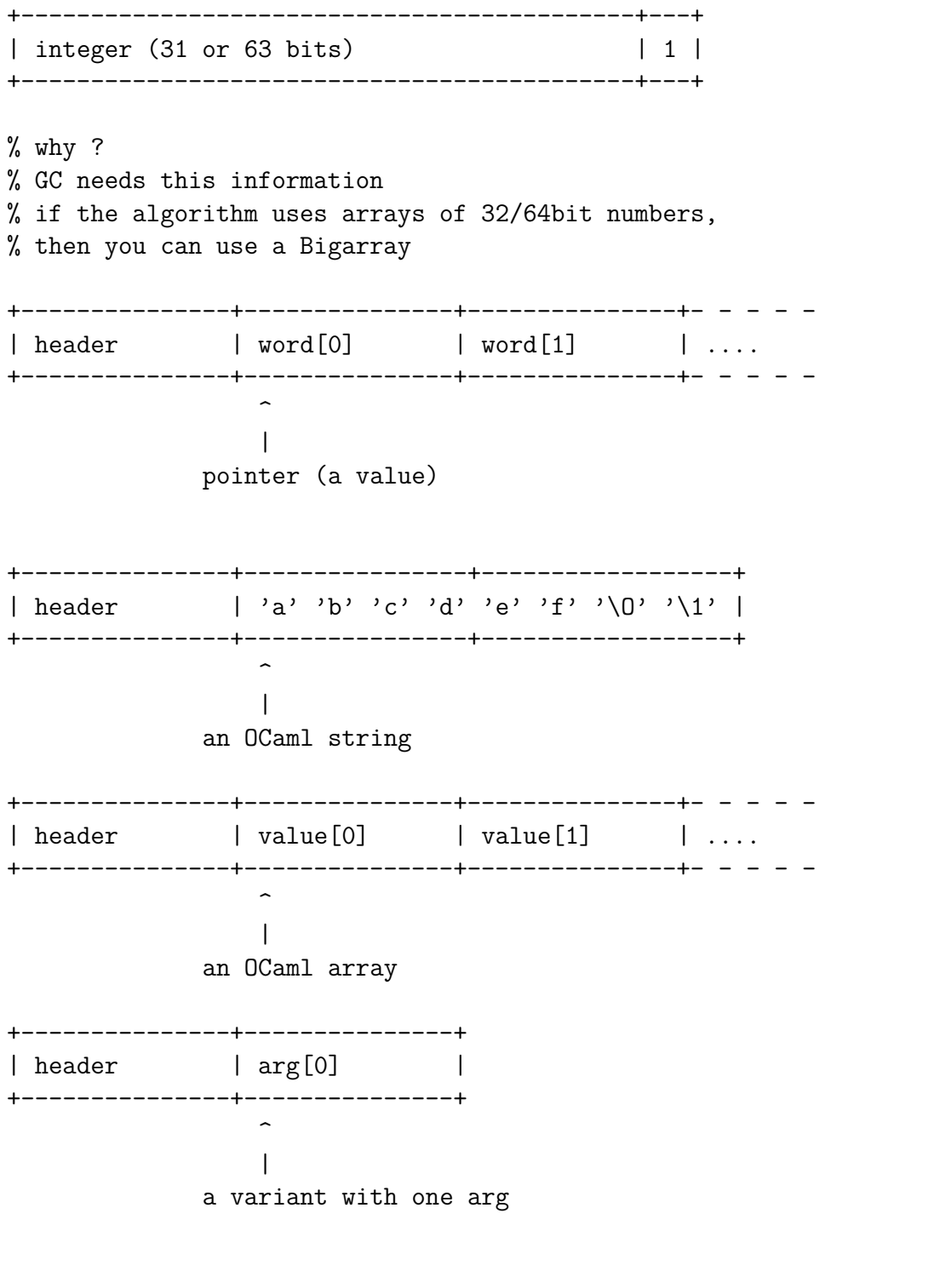
    ```
    module Enum = struct
      include Enum include Labels include Exceptionless
    end
    ```

    floating nested modules up (Enum.include, etc) include Enum, will expose all Enum have to the following context, so Enum.Labels is as Labels, so you can now include Labels, but *Labels.v will override Enum.v*, maybe you want it, and *module Enum still has Enum.Labels.v*, we just duplicated the nested module into toplevel

# 3 Runtime

- values
  integer-like (int,char,true, false, [], (), and some variants) (batteries dump) pointer (word-aligned, the bottom 2 bits of every pointer always 00, 3 bits 000 for 64-bit)

```
% 32 bit
+---------------------------------------+---+---+
| pointer                               | 0 | 0 |
+---------------------------------------+---+---+
```


integer (31 or 63 bits) 1
% why ?
% GC needs this information
% if the algorithm uses arrays of 32/64bit numbers,
% then you can use a Bigarray
header word[0] word[1] ....
pointer (a value)
header 'a' 'b' 'c' 'd' 'e' 'f' '\0' '\1'
an OCaml string
header value[0] value[1] ....
an OCaml array
header arg[0]
a variant with one arg
size of the block in words col tag byte
<- 2b-><--- 8 bits --->

```
offset -4 or -8
% 32 platform, it's 22bits long : the reason for the annoying 16MByte limit
% for string
% the tag byte is multipurpose
% in the variant-with-parameter example above, it tells you which
% variant it is. In the string case, it contains a little bit of runtime
% type information. In other cases it can tell the gc that it's a lazy value
% or opaque data that the gc should not scan



+---------------+---------------+---------------+- - - - -
| header        | float[0]                      | ....
+---------------+---------------+---------------+- - - - -
                  ^
                  |
              an OCaml float array

% in the file <byterun/mlvalues.h>
```

| | |
|---|---|
| any int, char | stored directly as a value, shifted left by 1 bit, with LSB=1 |
| (), [], false | stored as OCaml int 0 (native 1) |
| true | stored as OCaml int 1 |
| variant type t = Foo \| Bar \| Baz (no parameters) | stored as OCaml int 0,1,2 |
| variant type t = Foo \| Bar of int<br><br>list [1;2;3] | the varient with no parameters are stored as OCaml int 0,1,2, etc. counting just the variants that have no parameters. The variants with parameters are stored as blocks, counting just the variants with parameters. The parameteres are stored as words in the block itself. Note there is a limit around **240 variants with parameters that applies to each type**, but no limit on the number of variants without parameters yuou can have. **this limit arises because of the size of the tag byte and the fact that some of high numbered tags are resserved**<br>This is represented as 1::2::3::[] where [] is a value in OCaml int 0, and h::t is a block with tag 0 and two parameters. This representation is exactly the same as if list was a variant |
| tuples, struct and array | These are all represented identically, as a simple array of values, the tag is 0. The only difference is that an array can be allocated with variable size, but structs and tuples always have a fixed size. |
| struct or array where every elements is a float | These are treated as a special case. The tag has speical value `Dyn_array_tag` (254) so that the GC knows how to deal with these. **Note this exception does not apply to tuples that contains floats, beware anyone who would declare a vector as (1.0,2.0)**. |
| any string | strings are byte arrays in OCaml, but they have quite a clever representation to make it very efficient to get their length, and at the same time make them directly compatible with C strings. The tag is `String_tag` (252).<br><br>`Obj.("gshogh" \|> repr \|> tag);;`<br>`- : int = 252` |

Obj module

```
let a = [|1;2;3|] in Obj.(a|>repr|>tag);;
- : int = 0
Obj.(a |> repr |> size);;
- : int = 3

-- string has a clever algorithm
Obj.("ghsoghoshgoshgoshgoshogh"|> repr |> size);;
- : int = 4 (4*8 = 32 )
```

```
"ghsoghoshgoshgoshgoshogh" |> String.length;;
24 (padding 8 bits)
like all heap blocks, strings contain a header defining
the size of the string in  machine words.


("aaaaaaaaaaaaaaaa"|>String.length);;
- : int = 16
# Obj.("aaaaaaaaaaaaaaaa"|>repr |> size);;
- : int = 3

padding will tell you how many words are padded actually
number_of_words_in_block * sizeof(word) + last_byte_of_block - 1

The null-termination comes handy when passing a string to C, but is
not relied upon to compute the length (in Caml), allowing the string
to contain nulls.


repr : 'a -> t (id)
obj : t -> 'a (id)
magic : 'a -> 'b (id)

is_block : t -> bool = "caml_obj_is_block"
is_int : t -> bool = "%obj_is_int"

tag : t -> int ="caml_obj_tag" % get the tag field
set_tag : t -> int -> unit = "caml_obj_set_tag"

size : t -> int = "%obj_size" % get the size field

field : t -> int -> t = "%obj_field" % handle the array part
set_field : t -> int -> t -> unit = "%obj_set_field"

double_field : t -> int -> float
set_double_field : t -> int -> float -> unit

new_block : int -> int -> t = "caml_obj_block"

dup : t -> t = "caml_obj_dup"

truncate : t -> int -> unit = "caml_obj_truncate"
add_offset : t -> Int32.t -> t = "caml_obj_add_offset"

marshal : t -> string
```

```
Obj.(None |> repr |> is_int);;
- : bool = true

Obj.("ghsogho" |> repr |> is_block);;
- : bool = true

Obj.(let f x = x |> repr |> is_block in (f Bar, f (Baz 3)));;
- : bool * bool = (false, true)
```

## 3.1 GC

- heap
  Most OCaml blocks are created in the minor(young) heap.
  - minor heap (32K-words by default32 bit, 64K for 64bit)

    ```
    -- in my mac
    ledit ocaml -init x (avoid loading libraryes)
    Gc.stat ()
    78188 lsr 16 -> 1
    ```

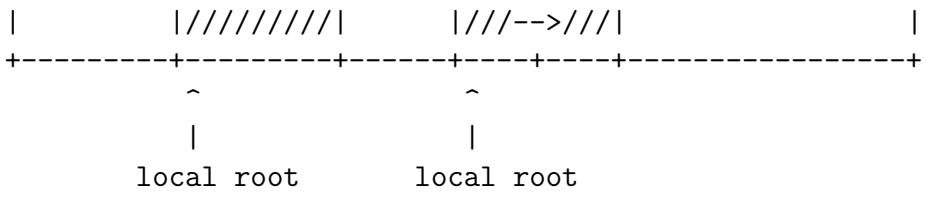
    ```
    +-----------------------------------------------------------+
    | unallocated                   |///allocated part/////////|
    +-----------------------------------------------------------+
     ^                                ^
     |                                |
    caml_young_limit              caml_young_ptr
                           <----- allocation proceeds
                                 in this direction
    ```

    Consider the array of two elments, the total size of this object will be 3 words (header + 2 words), so 24 bytes for 64-bit , so the fast path for allocation is

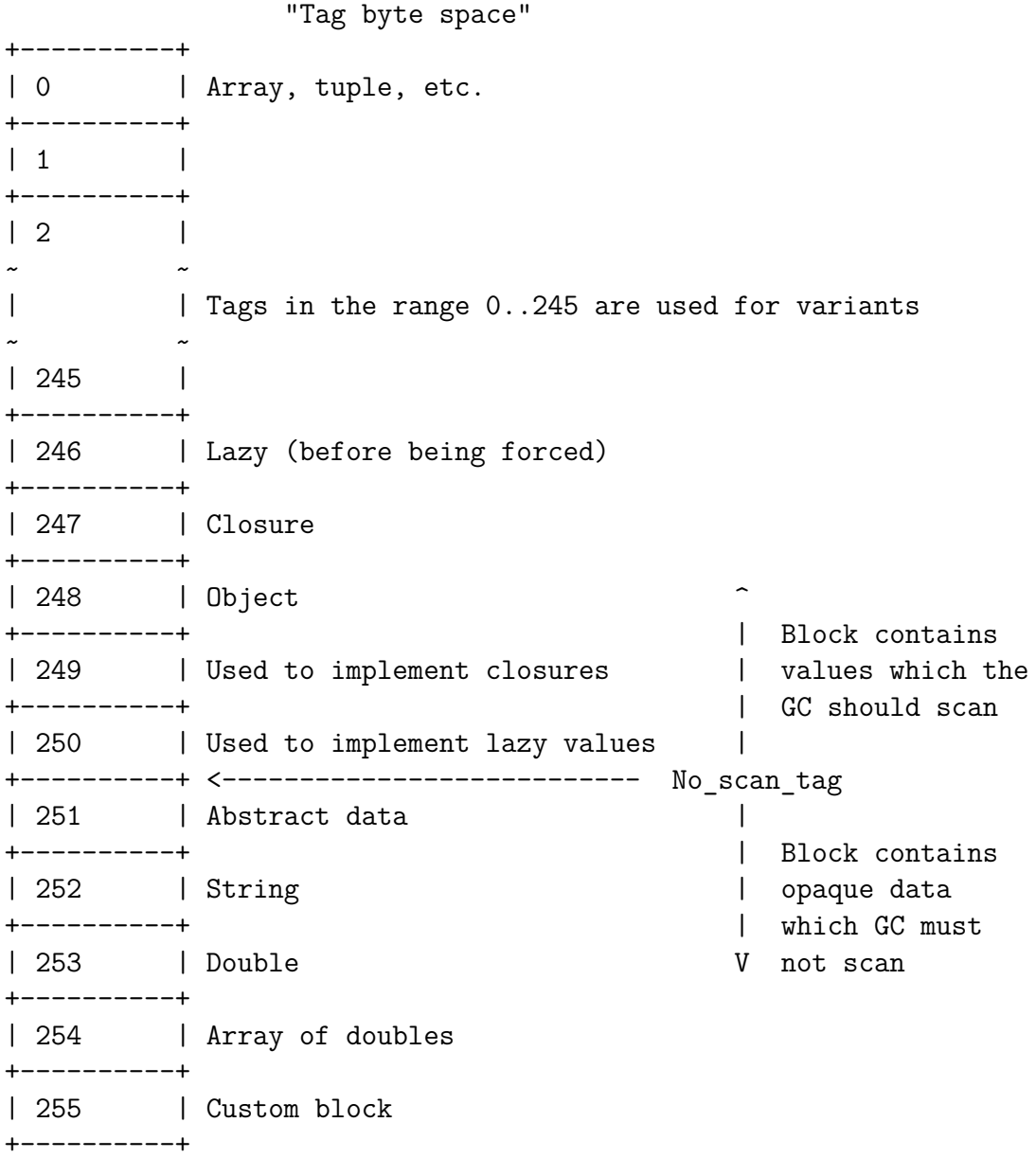
    ```
    Subtract size from caml_young_ptr.
    If caml_young_ptr < caml_young_limit, then take the slow path through the garba
    ```

    The fast path just **five machine instructions and no branches**. But even five instructions are costly in inner loops, be careful.
  - major heap
    when the minor heap runs out, it triggers a **minor collection**. The minor collection starts at all the local roots and "oldifies" them, bascially copies them by reallocating those objects (recursively) **to the major heap**. After this, any objects left in the minor heap **are unreachable**, so the minor heap can be reused by resetting `caml_young_ptr` .

    ```
               reachable        reachable
    +---------+---------+------+----+----+-----------------+
    ```




At runtime the garbage collector always knows what is a pointer, and what is an int or opaque data (like a string). Pointers get scanned so the GC can find unreachable blocks. Ints and opaque data must not be scanned. This is the reason for having a tag bit for integer-like values, and one of the uses of the tag byte in the header.



so, in the noral course of events, a small, long-lived object will start on the minor heap and be copied into the major heap. **Large objects go straight to the major heap**

But there is another important structure used in the major heap, called the **page table**. The garbage collector must at all times know which pieces of memory belong to the major heap, and which pieces of memory do not, and it uses the page table to track this.

One reason **why we always want to know where the major heap lies** is so we can avoid scanning pointers which point to C structs outside the OCaml heap. The GC will not stray beyond its own heap, and treats all pointers outside as opaque (it doesn't touch them or follow them).

In OCaml 3.10 the page table was implemented as a simple bitmap, with 1 bit per page of virtual memory (major heap chunks are always page-aligned). This was unsustainable for 64 bit address spaces where memory allocations can be very very **far apart**, so in OCaml 3.11 this was changed to a sparse hash table.

Because of the page table, C pointers can be stored directly as values, which saves time and space. (However, if your C pointer later gets freed, you must NULL the value-the reason is that the same memory address might later get malloced for the OCaml major heap, thus "suddenly" becoming a "valid" address agian. THIS usually results in crash ).

In a functional language **which does not allow any mutable references**, there's one guarantee you can make which is there could **never be a pointer going from the major heap to something in the minor heap**, so when an object in an immutable language graduates from the minor heap to the major heap, it is fixed forever(until it becomes unreachable), and can not point back to the minor heap.

But ocaml is impure, so if the minor heap collection worked exactly as previous, then the outcome wouldn't be good, maybe some object is not pointed at **by any local root**, so it would be "unreachable" and would "disappear", leaving a **dangling pointer**.

**one solution would be to check the major heap, but that would be massively time-consuming: minor-collections are supposed to be very quick** What OCaml does instead is to have a separate "refs" list. This contains a list of pointers that point **from the major heap to the minor heap**. During a minor heap collection, the refs list is consulted for additional roots(and after the minor heap collection, the refs list can be started anew).

The refs list however has to be updated, and it gets **updated potentially every time we modify a mutable field in a struct**. The code calls the c function `caml_modify` which both mutates the struct a nd decides whether this is a major→minor pointer to be added to the refs list.

If you use mutable fields then this is **much slower** than a simple assignment. However, **mutable integers** are ok, and don't trigger the extra call. You can also **mutate fields** yourself, eg. from c functions or using Obj, **provied you can**

**guarantee that this won't generate a pointer between the major and minor heaps**.

The OCaml gc does not collect the major heap in one go. It spreads the work over small **slices**, and splices are grouped into whole *phases* of work.

*A slice* is just a defined amount of work.

The phases are mark and sweep, and some additional subpasses dealing with weak pointers and finalization.

Finally there is *a compaction phase* which is triggered when there is no other work to do and the estimate of free space in the heap has reached some threshold. This is tunable. You can schedule when to compact the heap – while waiting for a keypress or between frames in a live simulation.

There is also a penalty for doing a slice of the major heap – for example if the minor heap is exhausted, then some activity in the major heap is unavoidable. However if you make the **minor heap large enough**, you can completely control when GC work is done. You can also move *large structures out of the major heap entirely*,

- module Gc

```
Gc.compact () ;;

let checkpoint p =
Gc.compact () ; prerr_endline ("checkpoint at
poisition " ^ p )

{-
the checkpoint function does two things:
Gc.compact () does a full major round of garbage
collection and compacts the heap. This is
the most aggressive form of Gc available, and it's
highly likely to segfault if the heap is corrupted.

prerr_endline prints a message to stderr and crucially
also flushes stderr, so you will see the message printed
immediately.
-}
```

grep for `caml_heap_check` in byterun for details

```
void caml_compact_heap (void)
{
  char *ch, *chend;
                                          Assert (caml_gc_phase == Phase_idle);
```

```
  caml_gc_message (0x10, "Compacting heap...\n", 0);

#ifdef DEBUG
  caml_heap_check ();
#endif


#ifdef DEBUG
void caml_heap_check (void)
{
  heap_stats (0);
}
#endif


#ifdef DEBUG
  ++ major_gc_counter;
  caml_heap_check ();
#endif
```

- tune
  problems can arise when you're building up ephemeral data structures which are larger than the minor heap. The data structure won't stay around overly long, but it is a bit too large. Triggering major GC slices more often can cause static data to be walked and re-walked more often than is necessary. tuning sample

```
let _ =
  let gc = Gc.get () in
    gc.Gc.max_overhead <- 1000000;
    gc.Gc.space_overhead <- 500;
    gc.Gc.major_heap_increment <- 10_000_000;
    gc.Gc.minor_heap_size <- 10_000_000;
    Gc.set gc
```

## 3.2 ocamlrun

- ocamlrun
  the ocamlrun command comprises three main parts: the bytecode interpreter, the memory allocator and garbage collector, and a set of c functions that implement primitive operations such as input/output.

# 4 Book

## 4.1 Developing Appilcations with Objective Caml

details

- caveat
  - + (modulo the boundary, *will not be checked*)
  - $1.0/0.0 \rightarrow \infty$
  - $+. - . * ./. * *modceilfloorsqrtexploglog10cossintanacosasinatan$
  - $asin3.14 \rightarrow nan$
  - `char_of_int 255` → `'\255'` (can not display)
  - `char_of_int int_of_char string_of_int int_of_string string_of_int 2551 -> ``2551`
  - string (length $\leq 2^{24} - 6$ )
  - == (physical equal) (=, != <>) for phisical equality

```
true == true;;
- : bool = true
# 3 == 3;;
- : bool = true
# 1. == 1.;;
- : bool = false
```

  - int * int * int is different from (int * int ) * int
  - unreasonable parametrical equality `(=) : 'a -> 'a -> bool`
  - recursive declaration

```
let rec ones = 1 :: ones;;
val ones : int list =
  [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   ...]
let special_size l =
```

```
    let rec size_aux prev = function
      |[] -> 0
      |_ :: l1  -> if List.memq l1 prev then 1 else 1 + size_aux (l1::prev) l1
      val special_size : 'a list -> int = <fun>
# special_size ones;;
- : int = 1
# let rec twos = 1 :: 2 :: twos in special_size twos;;
- : int = 2
# special_size [];;
- : int = 0

#
```

– combine patterns

```
p1 | .. |  pn (all name is forbidden within these patterns)
'a' .. 'e'

let test 'a' .. 'e' = true;;
           ^^^^^^^^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
'f'
val test : char -> bool = <fun>
```

– records

```
 type complex = {re:float;img:float};;
type complex = { re : float; img : float; }
# let add {re; img} {re; img} = 3;;
val add : complex -> complex -> int = <fun>
# let add {re; img} {re; img} = {re = re + re; img = img + img};;
Characters 36-38:
  let add {re; img} {re; img} = {re = re + re; img = img + img};;
```

– *redefinition marsks the previous one, while values of the maskeed types still exist, but it now turns to be an abstract type*

– excpetion

  * `Match_failure Division_by_zero Failure`
  * exception Name of t – monomorphic , extensible sum Type
    when pattern match your exception, its type should be fixed
  * control flow

– **disagree over interface**
when toplevel loads the same module (only the name is the same), it will check the interface is equal, this sucks since ocaml has flat namespace for module

- sharing
  for structured values, it will be sharing , however, *vectors of floats don't share*

```
let a = Array.create 3 0.;;
val a : float array = [|0.; 0.; 0.|]
# a.(0)==a.(1);;
- : bool = false
```

- weak type variables

```
    let b = ref [] -- b should '_a list ref
    -- since b is not pure, cannot be shared
    let a = [] -- a : 'a list
    let a = None -- a : 'a option
    let a = Array.create 3 None -- '_a option array
# type ('a,'b) t ={ch1 : 'a list; mutable ch2 : 'b list};;
type ('a, 'b) t = { ch1 : 'a list; mutable ch2 : 'b list; }
# let v = {ch1=[];ch2=[]};;
val v : ('a, '_b) t = {ch1 = []; ch2 = []}
    -- mutable sharing conflicts with polymorphism
```

- library
  - List

```
      @ length hd tl nth rev append rev_append concat flatten
      iter map rev_map left_fold fold_right iter2 map2 rev_map2
      fold_left2 fold_right2 for_all exists for_all2 exists2
      mem memq find filter partition assoc assq remove_assoc remove_assq
      split combine sort statble_sort fast_sort merge
# List.assq 3 [3,4;1,2];;
- : int = 4
# List.assq 3. [3.,4;1.,2];;
Exception: Not_found.
```

  - Array
    `Array.create_matrix creates Non-Rectangular matrices`

```
length get set make create init -- when you don't want to initialize
make_matrix (int->int->'a -> 'a array array) create_matrix;
append concat sub copy fill ('a array -> int -> int -> 'a -> int)
blit (Array.Labels.blit), to_list, of_list map iteri mapi fold_left
fold_right sort stable_sort fast_sort unsafe_get unsafe_set copy
```

  - IO

```
open_in open_out close_in close_out input_line
input (- : Batteries.Legacy.in_channel -> string -> int -> int -> int = <fun>)
output (- : Batteries.Legacy.out_channel -> string -> int -> int -> unit = <fun
read_line print_string print_newline print_endline
```

– stack (imperative data structure actually)

```
exceptin Empty
create
(*
type 'a t = { mutable c : 'a list }, mutable to delay initialization
*)
push pop top clear copy is_empty length iter enum copy
of_enum print
(module Exceptionless
top : 'a t -> 'a option, pop)
```

– stream

```
'a t
exception Failure
exception Error of string
from
of_list of_string of_channel iter empty peek junk count npeek
iapp icons ising lapp lcons lsing
sempty slazy dump npeek
```

syntax extension (don't use it in toplevel)

```
let concat_stream a b = [<a;b>]
-- expression not preceded by an ‘ considered to be sub-stream
```

destructive pattern matching (camlp5 can merge) consumed (error), failure use ocamllex ocamlyacc when you need it (consider it later)

– Array List String Hashtbl Buffer Queue

– Sort

```
module X = Sort ;;
module X :
  sig
    val list : ('a -> 'a -> bool) -> 'a list -> 'a list
    val array : ('a -> 'a -> bool) -> 'a array -> unit
    val merge : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list
  end
```

– Weak (vector of weak pointers) abstract type

```
sig
  type 'a t = 'a Weak.t
end
```

– Printf

```
%t -> (output->unit)
%t%s -> (output->unit)->string->unit
-- they all should be processed at compile time
```

– Digest
hash functions return a fingerprint of their entry (reversible)

```
   val string : string -> t -- fingerprint of a string
   val file : string -> t -- fingerprint of a file
```

– Marshal estimate data size

```
type external_flag = No_sharing | Closures

let size x = x |> flip Marshal.to_string [] |> flip Marshal.data_size 0;;
val size : 'a -> int = <fun>
# size 3;;
- : int = 1
# size 3.;;
- : int = 9
# size "ghsogho";;
- : int = 8
# size "ghsogho1";;
- : int = 9
# size "ghsogho1ah";;
- : int = 11
# size 111;;
- : int = 2
```

– Sys

```
os_type interactive word_size max_string_length
max_array_length time argv getenv command file_exists
remove rename chdir getcwd

# float (Sys.max_string_length ) /. (2. ** 57.);;
- : float = 0.999999999999999889
```

– Arg Filename Printexc

– Printexc

```
# module P = Printexc;;
module P :
  sig
    val to_string : exn -> string
    val catch : ('a -> 'b) -> 'a -> 'b
    val get_backtrace : unit -> string
    val record_backtrace : bool -> unit
    val backtrace_status : unit -> bool
    val register_printer : (exn -> string option) -> unit
    val pass : ('a -> 'b) -> 'a -> 'b
    val print : 'a BatInnerIO.output -> exn -> unit
    val print_backtrace : 'a BatInnerIO.output -> unit
  end
```

- Num
- `Arith_status`

```
# module X = Arith_status;;
module X :
  sig
    val arith_status : unit -> unit
    val get_error_when_null_denominator : unit -> bool
    val set_error_when_null_denominator : bool -> unit
    val get_normalize_ratio : unit -> bool
    val set_normalize_ratio : bool -> unit
    val get_normalize_ratio_when_printing : unit -> bool
    val set_normalize_ratio_when_printing : bool -> unit
    val get_approx_printing : unit -> bool
    val set_approx_printing : bool -> unit
    val get_floating_precision : unit -> int
    val set_floating_precision : int -> unit
  end
```

- Dynlink
  choice at exceution time, load a new module and hide the code code (hotpatch) actually (#load is kinda hotpatch), however to write it in programs *more flexible* than #load , load requries its name are fixed, and load will check .mli file, Dynlink does not do this check, while when you want to do X.blabla, it still checks, so still don't work, only side effects will work.

  ```
  #direcotry "+dynlink";;
  #load "dynlink.cma";;
  Dynlink.loadfile "test.cmo";;
  ```

- syntaxes

- expr

```
exp ::=value-path  -- value-name or module-path.value-name
  | constant
  | ( expr )
  | begin expr end
  | ( expr :  typexpr )
  | expr ,  expr  { , expr } -- tuple
  | constr  expr  -- constructor
  | ‘tag-name  expr  -- polymorphic variant
  | expr ::  expr  -- list
  | [ expr  { ; expr } ]
  | [| expr  { ; expr } |]
  | { field =  expr  { ; field =  expr } }
  | { expr with  field =  expr  { ; field =  expr } }
  | expr  { argument }+ -- application
  | prefix-symbol  expr  -- prefix operator
  | expr  infix-op  expr
  | expr .  field
  | expr .  field <-  expr  -- still an expression
  | expr .(  expr )
  | expr .(  expr ) <-  expr
  | expr .[  expr ]
  | expr .[  expr ] <-  expr
  | if expr then  expr  [ else expr ]
  | while expr do  expr done
  | for ident =  expr  ( to |  downto ) expr do  expr done
  | expr ;  expr
  | match expr with  pattern-matching
  | function pattern-matching
  | fun multiple-matching  -- multiple parameters matching
  | try expr with  pattern-matching
  | let [rec] let-binding   { and let-binding } in  expr
  | new class-path
  | object class-body end
  | expr #  method-name
  | inst-var-name
  | inst-var-name <-  expr
  | ( expr :>  typexpr )
  | ( expr :  typexpr :>  typexpr )
  | {< inst-var-name =  expr  { ; inst-var-name =  expr } >}
  | assert expr
  | lazy expr

argument::=expr
```

```
  | ~ label-name
  | ~ label-name :  expr
  | ? label-name
  | ? label-name :  expr

pattern-matching::=
 [|] pattern [when expr]-> expr { |pattern  [when expr] ->  expr }

multiple-matching::= { parameter }+  [when expr]-> expr

let-binding::=pattern =  expr
  | value-name  { parameter }  [: typexpr] =  expr

parameter::=pattern
  | ~ label-name
  | ~ ( label-name  [: typexpr] )
  | ~ label-name :  pattern
  | ? label-name
  | ? ( label-name  [: typexpr]  [= expr] )
  | ? label-name :  pattern
  | ? label-name : (  pattern  [: typexpr]  [= expr] )

##
  let f ?test:(Some x ) y = x + y;;
        ^^^^^^^^^^^^^^^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
None
val f : ?test:int -> int -> int = <fun>
##
```

- pattern

```
pattern ::= value-name
  | _
  | constant
  | pattern as  value-name
  | ( pattern )
  | ( pattern :  typexpr )
  | pattern |  pattern
  | constr  pattern
  | ‘tag-name  pattern
  | #typeconstr-name  -- object ?
  | pattern  { , pattern }
  | { field =  pattern  { ; field =  pattern } }
```

```
  | [ pattern  { ; pattern } ]
  | pattern ::  pattern
  | [| pattern  { ; pattern } |]
  | lazy pattern
```

- toplevel-phrase

```
toplevel-input::= { toplevel-phrase } ;;

toplevel-phrase::=definition
  | expr
  | #ident  directive-argument

directive-argument::=epsilon
  | string-literal
  | integer-literal
  | value-path
defition ::= let [rec] let-binding {and let-binding}
        | external value-name : typexpr = external-declartion
        | type-definition
        | exception-defition
        | class-definition
        | classtype-definition
        | module module-name {(module-name : module-type)} [:module-type] = module-
        | module type module-name = module-type
        | open module-path
        | include module-expr
```

- type-definition

```
type-definition ::= type typedef  { and typedef }

typedef ::= [type-params]  typeconstr-name  [type-information]

type-information::=
  [type-equation] [type-representation]{ type-constraint }
type-equation::= = typexpr

type-representation::=
          = constr-decl  { | constr-decl }
  | = { field-decl  { ; field-decl } }

type-params::= type-param
  | ( type-param  { , type-param } )
```

```
type-param::= ' ident
  | + ' ident
  | - ' ident

constr-decl::= constr-name
  | constr-name of  typexpr  { * typexpr }

field-decl::= field-name :  poly-typexpr
  | mutable field-name :  poly-typexpr
type-constraint ::=constraint ' ident =  typexpr

# type t;;
type t
```

## 4.2 Ocaml for scientists

- caveat
  - string char `'a' = '\097' "Hello world".[4]`

    `[|1;2;3|].(1) -- 2`
  - objects

```
-- it's a type class type
class type number = object
  method im:float
  method re:float
end

-- object
class complex x y = object
    val x = x
    val y = y
    method re:float = x
    method im:float = y
end ;;

let b : number = new complex 3. 4.

# let b = new complex 3. 4.;;
val b : complex = <obj>
# let b : number = new complex 3. 4.;;
val b : number = <obj>
```

```
-- immediate objects
# let make_z x y = object
    val x : float = x
    val y : float = y
    method re = x
    method im = y
    end;;
val make_z : float -> float -> < im : float; re : float > = <fun>

-- class type is kinda interface
# let abs_number (z:number) =
       let sqr x = x *. x in
       sqrt (sqr z#re +. sqr z#im);;

-- think class as a module
```

  - asr (arith) (**) lsr
  - elements

    ```
    [1;2;3;4] |> Set.of_list |> Set.elements;;
    ```

- convention
- GMP (GNU library for arbitrary precision arithmetic)

```
module type INT_RANGE = sig
type t
val make : int -> int -> t
end
```

- Hashtbl(create, Make) Hahsing is another form of structural comparison and should not be applied **to abstract types** *Semantically equivalent sets are likely to produce different hashes* notice Map.empty is polymorphic, Hashtbl.empty is monomorphic

## 4.3 OCaml introduction book-ml

## 4.4 caltech ocaml book

- oo
  - immediate object

    ```
    let poly = object
      val vertices = [|0,0;1,1;2,2|]
      method draw = "test"
    end
    ```

– dynamic lookup
obj#method, the actual method that gets called is determined at *runtime*

```
# let draw_list items = List.iter (fun item->item#draw) items;;
val draw_list : < draw : unit; .. > list -> unit = <fun>
```

– type annotation (very common in oo)

– .. ellipse – row variable
<> represents a functional update (only fields), which produces a new object

```
# type 'a blob = <draw : unit; ..> as 'a ;;
type 'a blob = 'a constraint 'a = < draw : unit; .. >

let transform =
    object
      val matrix = (1.,0.,0.,0.,1.,0.)
      method new_scale sx sy =
        {<matrix= (sx,0.,0.,0.,sy,0.)>}
      method new_rotate theta =
        let s,c=sin theta, cos theta in
        {<matrix=(c,-.s,0.,s,c,0.)>}
      method new_translate dx dy=
        {<matrix=(1.,0.,dx,0.,1.,dy)>}
      method transform (x,y) =
        let (m11,m12,m13,m21,m22,m23)=matrix in
        (m11 *. x +. m12 *. y +. m13,
         m21 *. x +. m22 *. y +. m23)
    end ;;

                              val transform :
  < new_rotate : float -> 'a; new_scale : float -> float -> 'a;
    new_translate : float -> float -> 'a;
    transform : float * float -> float * float >
  as 'a = <obj>

#   let new_collection () =
    object
      val mutable items = []
      method add item = items <- item::items
      method transform mat =
        {<items = List.map (fun item -> item#transform mat) items>}
    end ;;

              val new_collection :
  unit ->
  (< add : (< transform : 'c -> 'b; .. > as 'b) -> unit;
```

```
  transform : 'c -> 'a >
 as 'a) =
<fun>
```

- caveat
  - field expression may not refer to other fields, nor to self
  - after you get the object you can have initializer
    the object does not exist when the field values are be computed for the initializer, you can call `self#blabla`

    ```
    #   object
        val x = 1
        val mutable x_plus_1 = 0
        initializer
          x_plus_1 <- x + 1
    end ;;
    - : <  > = <obj>
    ```
  - method private
  - subtyping supports width and depth subtyping supports contravariant and covariant for subtyping of recursive object types, first assume it is right then prove it using such assumption

    ```
    e : t1 :> t2 -- sometimes type annotation and coersion both needed
    -- when t2 is recursive or t2 has polymorphic structure
    ```
  - narrowing (opposite to subtyping) (not permitted in Ocaml) but you can simulate it. do runtime type testing

    ```
    type animal = < eat : unit; v : exn >
    type dog = < bark : unit; eat : unit; v : exn >
    type cat = < eat : unit; meow : unit; v : exn >

    let fido : dog = object(self) method v=Dog self method eat = () method bark

    let miao : cat = object(self) method v = Cat self method eat = () method me

    -- so you dispatch on animal#v
    ```

    you can also encode using polymorphic variant sometimes ocaml's type annotation does not require its polymorphic is also a feature, you just hint, and let it guess

    ```
    type 'a animal = <eat:unit; tag : [>] as 'a >;;

    let fido : 'a animal = object method eat = () method tag = `Dog 3 end;;
    val fido : [> `Dog of int ] animal = <obj>
    -- ok
    ```

```
(*
# let fido : [< `Dog of int] animal = object method eat = () method tag = `
val fido : [ `Dog of int ] animal = <obj>
*)

let miao : [> `Cat of int] animal = object method eat = () method tag = `Cat
val miao : [> `Cat of int ] animal = <obj>
# [fido;miao];;
- : [> `Cat of int | `Dog of int ] animal list = [<obj>; <obj>]


List.map (fun v -> match v#tag with `Cat a -> a |`Dog a -> a) [fido;miao];;
- : int list = [3; 2]
```

* modules vs objects
  - objects (data entirely hidden)
  - now both are first class (both can be used as arguments)
  - objects can bind type variable easier, espeically when self recursive recursive is so natural in objects (isomorphic-like equivalence is free in oo) when we build an object of recursive type, but we don't care which type it is (maybe called existential type), so coding existential types is easier in OO

```
module type PolySig = sig
type poly
val create : (float*float) array -> poly
val draw : poly -> unit
val transform : poly -> poly
end
module Poly :PolySig =
type poly = (float * float) array
let create vertices = vertices
let draw vertices = ()
let transform matrix = matrix
end



-- here module Poly is more natural to model it as an object
# class type poly  = object
 method create : (float*float) array -> poly
 method draw : poly -> unit
 method transform : poly->poly
end
;;
class type poly =
```

```
  object
    method create : (float * float) array -> poly
    method draw : poly -> unit
    method transform : poly -> poly
  end
-- similar to module type
-- or immediate object
--
class poly = object (self:'self)
    method test (x:'self) = x end;;
  class poly : object ('a) method test : 'a -> 'a end
# let v = new poly;;

-- or
type blob = <draw:unit-> unit; transform:unit-> blob>;;
type blob = < draw : unit -> unit; transform : unit -> blob >
type blob = {draw:unit-> unit; transform:unit-> blob};;
```

  * parameterized class
    template shows how to build an object
  * polymoprhic class

```
class ['a] cell(x:'a) = object
  method get = x
end ;;
class ['a] cell : 'a -> object method get : 'a end
```

- polymorphic variants

  –

```
-- definition
-- closed
let string_of_number = function `Integer i -> i;;
val string_of_number : [< `Integer of 'a ] -> 'a = <fun>

-- open
# let string_of_number = function
    |`Integer i -> i
    |_ -> invalid_arg "string_of_number";;
val string_of_number : [> `Integer of 'a ] -> 'a = <fun>

let test0 = function
  |`Int i -> i

let test1 = function
  |`Int i -> i
  | _ -> invalid_arg "invalid arg in test1"
```

```
let test2 = function
  |x -> test0 x

let test3 = function
  |x -> test1 x

(* let test4 : [> `Real of 'a | `Int of 'a ] -> 'a = function *)
(*   |`Real x -> x *)
(*   | x -> test0 (x:> [< `Int of 'a]) *)

let test5 = function
  |`Real x -> x
  | x -> test1 x


val test0 : [< `Int of 'a ] -> 'a = <fun>
val test1 : [> `Int of 'a ] -> 'a = <fun>
val test2 : [< `Int of 'a ] -> 'a = <fun>
val test3 : [> `Int of 'a ] -> 'a = <fun>
val test5 : [> `Int of 'a | `Real of 'a ] -> 'a = <fun>

(** for open union, it's easy to reuse, but unsafe
    for closed union, hard to use, since the type checker is
    conservative
*)

test1 `Test;;
Exception: Invalid_argument "invalid arg in test1".

test0 `Test;;
Characters 6-11:
  test0 `Test;;
        ^^^^^
Error: This expression has type [> `Test ]
       but an expression was expected of type [< `Int of 'a ]
       The second variant type does not allow tag(s) `Test
```

– define

```
type number = [> `Integer of int | `Real of float ];;
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: A type variable is unbound in this type declaration.
```

```
In type [> `Integer of int | `Real of float ] as 'a
the variable 'a is unbound

type 'a number = 'a constraint 'a = [>`Integer of int | `Real of float]

let zero : 'a number = `Zero;;
val zero : [> `Integer of int | `Real of float | `Zero ] number = `Zero


type number = [< `Integer of int | `Real of float ];;
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: A type variable is unbound in this type declaration.
In type [< `Integer of int | `Real of float ] as 'a
the variable 'a is unbound
# type number = [ `Integer of int | `Real of float ];;
type number = [ `Integer of int | `Real of float ]
```

– subtyping

```
[`A] :> [`A | `B]
-- because you know how to handle A and B, then you know
-- how to handle A

let f x = (x:[`A] :> [`A | `B ]);;
val f : [ `A ] -> [ `A | `B ] = <fun>

-- ocaml does has width and depth subtyping
if t1 :> t1' and t2 :> t2' then (t1,t2) :> (t1',t2')

let f x = (x:[`A] * [`B] :> [`A|`C] * [`B | `D]);;
val f : [ `A ] * [ `B ] -> [ `A | `C ] * [ `B | `D ] = <fun>

--
let f x = (x : [ `A | `B ] -> [ `C ] :> [ `A ] -> [ `C | `D ]);;
val f : ([ `A | `B ] -> [ `C ]) -> [ `A ] -> [ `C | `D ] = <fun>
```

– variance notation
if you don't write the + and -, ocaml will infer them for you , but when you write abstract type in module tpe signatures, it makes sense. variace annotations allow you to expose the subtyping properties of your type in an interface, without exposing the representation.

```
type (+'a, +'b) t = 'a * 'b
type (-'a,+'b) t = 'a -> 'b
```

```
module M : sig
  type (+'a,'+b) t
end = struct
  type ('a,'b) t = 'a * 'b
end
-- ocaml did the check when you define it
-- so you can not define it arbitrarily
```

– covariant helps polymorphism

```
module M : sig
    type +'a t
    val embed : 'a -> 'a t
  end = struct
    type 'a t = 'a
    let embed x = x
end ;;
M.embed []  ;;
- : 'a list M.t = <abstr>
```

– example

```
type suit = [ `Club | `Diamond | `Heart | `Spade ]
let winner = function `Heart -> true | #suit -> false;;
val winner : [< suit ] -> bool = <fun>

let winner2 = function `Unknown -> true |#suit -> false;;
val winner2 : [< `Club | `Diamond | `Heart | `Spade | `Unknown ] -> bool =
  <fun>

-- the variant tag does not belong to a particular type

let winner3 : (suit -> bool) = function `Unknown -> true | #suit -> false;;
                                        ^^^^^^^^

Warning 11: this match case is unused.
val winner3 : suit -> bool = <fun>
```

## 4.5 The functional approach to programming

## 4.6 tricks

- ocamlobjinfo
  analysing ocaml obj info

```
 ocamlobjinfo ./_build/src/batEnum.cmo
File ./_build/src/batEnum.cmo
```

```
Unit name: BatEnum
Interfaces imported:
720848e0b508273805ef38d884a57618 Array
c91c0bbb9f7670b10cdc0f2dcc57c5f9 Int32
42fecddd710bb96856120e550f33050d BatEnum
d1bb48f7b061c10756e8a5823ef6d2eb BatInterfaces
81da2f450287aeff11718936b0cb4546 BatValue_printer
6fdd8205a679c3020487ba2f941930bb BatInnerIO
40bf652f22a33a7cfa05ee1dd5e0d7e4 Buffer
c02313bdd8cc849d89fa24b024366726 BatConcurrent
3dee29b414dd26a1cfca3bbdf20e7dfc Char
db723a1798b122e08919a2bfed062514 Pervasives
227fb38c6dfc5c0f1b050ee46651eebe CamlinternalLazy
9c85fb419d52a8fd876c84784374e0cf List
79fd3a55345b718296e878c0e7bed10e Queue
9cf8941f15489d84ebd11297f6b92182 CamlinternalOO
b64305dcc933950725d3137468a0e434 ArrayLabels
64339e3c28b4a17a8ec728e5f20a3cf6 BatRef
3aeb33d11433c95bb62053c65665eb76 Obj
3b0ed254d84078b0f21da765b10741e3 BatMonad
aaa46201460de222b812caf2f6636244 Lazy
Uses unsafe features: YES
Primitives declared in this module:
%identity
%identity


ocamlobjinfo /Users/bob/SourceCode/ML/godi/lib/ocaml/std-lib/camlp4/camlp4lib.cma |
Unit name: Camlp4_import
Unit name: Camlp4_config
Unit name: Camlp4
```

obj has many Units, each Unit itself also import some interfaces.

- operator associativity
  the first char decides

```
@ right
^ right
# let (^|) a b = a - b;;
val ( ^| ) : int -> int -> int = <fun>
# 3 ^| 2 ^| 1;;
- : int = 2
```

- literals

```
30l => int32
30L => int64
30n => nativeint
```

- `{re ;_}` some labels were intensionally omiteed

- emacs
  there are some many tricks I can only enum a few

  – capture the shell command

    ```
    C-u M-! to capture the shell-command
    M-| shell-command-on-region
    ```

- compiling

```
# let ic = Unix.open_process_in "ocamlc test.ml 2>&1";;
val ic : in_channel = <abstr>

# input_line ic;;
- : string = "File \"test.ml\", line 1, characters 0-1:"

# input_line ic;;
- : string = "Error: I/O error: test.ml: No such file or directory"

# input_line ic;;
Exception: End_of_file.
```

- toplevellib.cma (toplevel/toploop.mli)

- memory profiling

```
You can override a little ocaml-benchmark to measure the allocation rate
of the GC. This gives you a pretty good understanding on the fact you
are allocating too much or not.
```

```
(** Benchmark extension
   @author Sylvain Le Gall
 *)

open Benchmark;;

type t =
```

```
  {
    benchmark: Benchmark.t;
    memory_used: float;
  }
;;

let gc_wrap f x =
 (* Extend sample to add GC stat *)
 let add_gc_stat memory_used samples =
   List.map
     (fun (name, lst) ->
        name,
        List.map
          (fun bt ->
             {
               benchmark = bt;
               memory_used = memory_used;
             }
          )
          lst
     )
     samples
 in

 (* Call throughput1 and add GC stat *)
 let () =
   print_string "Cleaning memory before benchmark"; print_newline ();
   Gc.full_major ()
 in
 let allocated_before =
   Gc.allocated_bytes ()
 in
 let samples =
   f x
 in
 let () =
   print_string "Cleaning memory after benchmark"; print_newline ();
   Gc.full_major ()
 in
 let memory_used =
   ((Gc.allocated_bytes ()) -. allocated_before)
 in
   add_gc_stat memory_used samples
;;
```

```
let throughput1
     ?min_count ?style
     ?fwidth    ?fdigits
     ?repeat    ?name
     seconds
     f x =

 (* Benchmark throughput1 as it should be called *)
 gc_wrap
   (throughput1
      ?min_count ?style
      ?fwidth    ?fdigits
      ?repeat    ?name
      seconds f) x
;;

let throughputN
     ?min_count ?style
     ?fwidth    ?fdigits
     ?repeat
     seconds name_f_args =
 List.flatten
   (List.map
      (fun (name, f, args) ->
        throughput1
          ?min_count ?style
          ?fwidth    ?fdigits
          ?repeat    ~name:name
          seconds f args)
      name_f_args)
;;

let latency1
     ?min_cpu ?style
     ?fwidth  ?fdigits
     ?repeat  n
     ?name    f x =
 gc_wrap
   (latency1
     ?min_cpu ?style
     ?fwidth  ?fdigits
     ?repeat  n
     ?name    f) x
;;
```

```
let latencyN
     ?min_cpu ?style
     ?fwidth  ?fdigits
     ?repeat
     n name_f_args =
 List.flatten
   (List.map
      (fun (name, f, args) ->
        latency1
          ?min_cpu   ?style
          ?fwidth    ?fdigits
          ?repeat    ~name:name
          n          f args)
      name_f_args)
;;
```

## 4.7 ocaml blogs

ygrek michal eigenclass syntax