OCaml Hacks

Hongbo Zhang

January 24, 2012

Preface

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

☺

Acknowledgements

write later

Contents

Todo list

Chapter 1

eco-system

1.1 ocamlbuild

The reason for ocamlbuild in OCaml is to solve the complex scheme to build camlp4. But it's very useful in other aspects

Your code is in the `_build` directory. `ocamlbuild` copies the `needed` source files and compiles them.

In `_build`, `_log` file logs detailed building process.

`ocamlbuild` automatically creates a symbol link to the executable it in the current directory

hygiene rules at start up (.cmo, .cmi, or .o should appear outside of the `_build`) (`-no-hygiene`)

Important Compile Falgs

option	comment
-quiet	
-verbose <level>	
-documentation	show rules and flags for a specific `_tags` file
-clean	
-r	Traverse directories by default(true:traverse)
-I <path>	
-Is <path,...>	
-X <path>	ignore directory
-Xs <path,...>	
-lib <flag>	link to ocaml library
-libs <flag,...>	
-mod <module>	link to ocaml module
-mods	
-pkg <package>	link to `ocaml findlib` package
-pkgs <...>	
-lflag <flag>	ocamlc link flags
-lflags	
-cflag	ocamlc comple flags
-cflags	
-yaccflag	
-yaccflags	
-lexflag	
-lexflags	
-pp	preprocessing flagss
-tag <tag>	add to default tags
-tags	
-show-tags	`ocamlbuild -show-tags target`
-ignore <module,...>	
-no-hygiene	
-no-plugin	
-just-plugin	just build myocamlbuild.ml
-use-menhir	
-use-jocaml	
-use-ocamlfild	
-build-dir	set build directory (implies no-links)
-install-lib-dir <path>	
-install-bin-dir	
-ocamlc <command>	set the ocamlc command
-ocamlopt	
-ocamldoc	
-ocamlyacc	
-menhir	set the menhir tool (use it after -use-menhir)
-ocamllex	
-ocamlmktop	
-ocamlrun	
--	supply arguments

Simple Examples

1. `ocamlbuild -quiet xx.native -- args`
2. `ocamlbuild -quite -use-ocamlfind xx.native -- args`
3. pass flags to `ocamlc` at compile timei.e. `-cflags -I,+lablgtk,-rectypes`
4. linking with **external** libraries. i.e. `-libs unix,num`. You may need add the options below to make it work if this not in OCaml's default search path

`-cflags -I,/usr/local/lib/ocaml -lflags -I,/usr/local/lib/ocaml`

5. `mllib` file

```
cat top_level.mllib
Dir_top_level_util
Dir_top_level
```

then you can `ocamlbuild top\_level.cma`, then you can use **ocamlobjinfo** to see exactly which modules are compacted into it.

```
ocamlobjinfo _build/top_level.cma | grep Unit
Unit name: Dir_top_level_util
Unit name: Dir_top_level
```

6. `mlpack` file hierachical packing

mlpack

`_tags` File
Every source has a set of tags .

```
bash-3.2$ocamlbuild -show-tags test.ml
Tags for "test.ml":
  {. extension:ml, file:test.ml, ocaml, pkg_camlp4.macro,
     pkg_menhirLib,
     pkg_ulex, predefine_ulex.ml, quiet, syntax_camlp4o, traverse,
        use_menhir .}
bash-3.2$ ocamlbuild -show-tags test.byte
Tags for "test.byte":
  {. byte, extension:byte, file:test.byte, ocaml, pkg_menhirLib,
     pkg_ulex,
     program, quiet, traverse, use_menhir .}
bash-3.2$ ocamlbuild -show-tags test.native
Tags for "test.native":
  {. extension:native, file:test.native, native, ocaml,
     pkg_menhirLib,
     pkg_ulex, program, quiet, traverse, use_menhir .}
```

By preceding a tag with a minus sign, one may `remove` tags from one or more files.

The built-in `_tags` file

```
<**/*.ml>   or <**/*.mli> or <**/*.ml.depends> : ocaml
<**/*.byte> : ocaml , byte , program
<**/*.native>: ocaml , native , program
<**/*.cma>:ocaml , byte ,library
<**/*.cmxa>:ocaml ,native ,library
<**/*.cmo>:ocaml ,byte
<**/*.cmx>:ocaml ,native
```

<**/*.ml> means that .ml files in *current dir or sub dir*. A special tag made from the path name of the file relative to the toplevel of the project is automatically defined for each file. Just as above `test.ml` will be tagged `file:test.ml`and also `extension:ml`

1. Including subdirectories `include <foo> or <bar> or <baz> : include` suppose you are calling `ocamlbuild foo/main.byte`, then use Foo, Foo.Bar and Foo.Baz in your code

2. grouping targets `foo.itarget, foo.otarget`

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

3. packing `foo.mlpack`

   ```
   cat foo.mlpack
   Bar
   Baz
   ```

4. grouping `foo.mllib`

5. preprocessing either `-pp` or tags `pp(cmd ...)`

6. debuggin and profiling either `.d.byte, .p.native` or `true:debug`

7. documentation target

   create a file called `foo.odocl`, then write the modules you want to document, then build the target `foo.docdir/index.html` when you use -keep-code flag

in myocamlbuild.ml, *only* document of exposed modules are kept, not very useful flag ["ocaml"; "doc"] & S[A"-keep-code"]; ocamldep seems to be **lightweight** ocamlbuild -ocamldoc 'ocamlfind ocamldoc -keep-code' foo.docdir/inc
It's weird when you have mli file, -keep-code does not work

8. glob patterns

glob patterns

With lex yacc, ocamlfind

1. .mll .mly supported by default, {menhir (-use-menhir) or add a line true : use_menhir

2. add a line in tags file <*.ml> : pkg_sexplib.syntax, pkg_batteries.syntax, syntax_can
   here syntax_camlp4o is translated by myocamlbuild.ml to -syntax camlp4o to pass to ocamlfind pkg needs **ocamlbuild plugin** support.

Examples with Syntax extension

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

```
<pa_*r.{ml,cmo,byte}> : pkg_dynlink , pp(camlp4rf ), use_camlp4_full

<*_ulex.{byte,native}> : pkg_ulex

<*_ulex.ml> : syntax_camlp4o,pkg_ulex,pkg_camlp4.macro

<*_r.ml>:syntax_camlp4r,pkg_camlp4.quotations.r,pkg_camlp4.macro,pkg_camlp4.extend

pa_vector_r.ml:syntax_camlp4r,pkg_camlp4.quotations.r,pkg_camlp4.extend,pkg_sexpli

<pa_vector_r.{cmo,byte,native}>:pkg_dynlink,use_camlp4_full,pkg_sexplib

<*_o.ml> : syntax_camlp4o,pkg_sexplib.syntax
```

```
"map_filter_r.ml" : pp(camlp4r -filter map)

"wiki_r.ml" or "wiki2_r.ml"  : pp(camlp4rf -filter meta), use_camlp4_full

"wiki2_r.mli" : use_camlp4_full
```

The .mli file also needs tags. For syntax extension, **order matters** , For more information, check out **camlp4/examples**. when you use pp flag, you need to specify the path to `pa_xx.cmo`, so symbol link may help. Since 3.12,, you can use `-use-ocamlfind` to activate. ocamlfind predicates can be activated with the `predicate(...)` tag.

```
  <*.ml>: package(lwt.unix), package(lwt.syntax), syntax(camlp4o)
"prog.byte": package(lwt.unix)
```

Interaction with git

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

ocamlbuild cares white space, **take care when write tags file**

Rules

A rule is composed of triple (Tags, Targets => Dependencies).

Principal

`ocamlbuild` looks for all rules that are valid for this target. You can set `-verbose 10` to get the backtrace in case of a failure.

Plugins Plugin API

There are 3 stages,(hygiene, options(parsing the command line options), rules(adding the default rules to the system)). You can add hooks to what you want.

`{Before|After}_{options|hygiene|rules}` To change the options, simply refer to the `Options` module.

```
sub_modules "Ocamlbuild_plugin";;
module This_module_name_should_not_be_used :
    module Pathname :
        module Operators :
    module Tags :
        module Operators :
    module Command :
    module Outcome :
    module String :
    module List :
    module StringSet :
    module Options :
    module Arch :
    module Findlib :
```

Useful API,

`Pathname.t`,`Tags.elt`string List the tags of a file `tags_of_pathname` Tag a file `tag_file` Untag a file `tag_file "x.ml" ["-use_unix"]` `Arch.print_info`

```
rule;;
- : string ->
    ?tags:string list ->
    ?prods:string list ->
    ?deps:string list ->
    ?prod:string ->
    ?dep:string ->
    ?stamp:string ->
    ?insert:[ `after of string | `before of string | `bottom | `top ] ->
    Ocamlbuild_plugin.action -> unit
= <fun>
```

The first arg is the name of the rule(unique required), `~dep` is the dependency, `~prod` is the production. For example with `~dep:"%.ml" ~prod:"%.byte"`, you can produce “bla.byte” from “bal.ml”. There are some predefined commands such as Unix commands(cp,mv,...).

flag,dep

```
flag ["ocaml"; "compile"; "thread"'] (A "-thread")
```

It says tags `ocaml, compile, thread` should become `-thread`

```
type t =
  |Seq of t list
  (* A sequence of commands (like the ‘;’ in shell) *)
  |Cmd of spec
  (* A command is made of command specifications (spec) *)
  |Echo of string list * pathname
  (* Write the given strings (w/ any formatting) to the given file *)
  |Nop
  (*The type t provides some basic combinators and command
    primitives. Other commands can be made of command specifications
    (spec).*)
type spec =
  |N (*No operation.        *)
  |S of spec list(* A sequence. This gets flattened in the last stages*)
  |A of string        (* An atom.*)
  |P of pathname(* A pathname.*)
  |Px of pathname
  (* A pathname, that will also be given to the call_with_target
     hook. *)
  |Sh of string
  (* A bit of raw shell code, that will not be escaped. *)
  |T of tags
  (* A set of tags, that describe properties and some semantics
     information about the command, afterward these tags will be replaced
```

```
    by command specs (flags for instance). *)
  |V of string
  (* A virtual command, that will be resolved at execution using
    resolve_virtuals *)
  |Quote of spec
  (* A string that should be quoted like a filename but isn't really
   one. *)
```

```
module Options
```

contains refs to be configured

```
module type OPTIONS = sig
  type command_spec

  val build_dir : string ref
  val include_dirs : string list ref
  val exclude_dirs : string list ref
  val nothing_should_be_rebuilt : bool ref
  val ocamlc : command_spec ref
  val ocamlopt : command_spec ref
  val ocamldep : command_spec ref
  val ocamldoc : command_spec ref
  val ocamlyacc : command_spec ref
  val ocamllex : command_spec ref
  val ocamlrun : command_spec ref
  val ocamlmklib : command_spec ref
  val ocamlmktop : command_spec ref
  val hygiene : bool ref
  val sanitize : bool ref
  val sanitization_script : string ref
  val ignore_auto : bool ref
  val plugin : bool ref
```

```
val just_plugin : bool ref
val native_plugin : bool ref
val make_links : bool ref
val nostdlib : bool ref
val program_to_execute : bool ref
val must_clean : bool ref
val catch_errors : bool ref
val use_menhir : bool ref
val show_documentation : bool ref
val recursive : bool ref
val use_ocamlfind : bool ref

val targets : string list ref
val ocaml_libs : string list ref
val ocaml_mods : string list ref
val ocaml_pkgs : string list ref
val ocaml_cflags : string list ref
val ocaml_lflags : string list ref
val ocaml_ppflags : string list ref
val ocaml_yaccflags : string list ref
val ocaml_lexflags : string list ref
val program_args : string list ref
val ignore_list : string list ref
val tags : string list ref
val tag_lines : string list ref
val show_tags : string list ref

val ext_obj : string ref
val ext_lib : string ref
val ext_dll : string ref
val exe : string ref
```

```
  val add : string * Arg.spec * string -> unit
end
```

Some Examples

```
open Ocamlbuild_plugin;;
open Command;;

let alphaCaml = A"alphaCaml";;

dispatch begin function
  | After_rules ->
      rule "alphaCaml: mla -> ml & mli"
        ~prods:["%.ml"; "%.mli"]
        ~dep:"%.mla"
      begin fun env _build ->
        Cmd(S[alphaCaml; P(env "%.mla")])
      end
  | _ -> ()
end
```

```
(* Open the ocamlbuild world... *)
open Ocamlbuild_plugin;;

(* We work with commands so often... *)
open Command;;

(* This dispatch call allows to control the execution order of your
   directives. *)
dispatch begin function
  (* Add our rules after the standard ones. *)
| After_rules ->
```

```
    (* Add pa_openin.cmo to the ocaml pre-processor when use_opening is set *)
    flag ["ocaml"; "pp"; "use_openin"] (A"pa_openin.cmo");

    (* Running ocamldep on ocaml code that is tagged with use_openin will requi
       Note that you only need this declaration when the syntax extension is pa
       sources to be compiled with ocamlbuild. *)
    dep ["ocaml"; "ocamldep"; "use_openin"] ["pa_openin.cmo"];
| _ -> ()
end;;
```

```
"bar.ml": camlp4o, use_openin
<foo/*.ml> or <baz/**/*.ml>: camlp4r, use_openin
"pa_openin.ml": use_camlp4, camlp4o
```

```
open Ocamlbuild_plugin
open Unix

let version = "1.4.2+dev"

let time =
  let tm = Unix.gmtime (Unix.time ()) in
  Printf.sprintf "%02d/%02d/%04d %02d:%02d:%02d UTC"
    (tm.tm_mon + 1) tm.tm_mday (tm.tm_year + 1900)
    tm.tm_hour tm.tm_min tm.tm_sec

let make_version _ _ =
  let cmd =
    Printf.sprintf "let version = %S\n\
                     let compile_time = %S"
      version time
```

```
  in
  Cmd (S [ A "echo"; Quote (Sh cmd); Sh ">"; P "version.ml" ])

let () = dispatch begin function
  | After_rules ->
      rule "version.ml" ~prod: "version.ml" make_version
  | _ -> ()
end

open Ocamlbuild_plugin

let () =
  dispatch begin function
  | After_rules ->
    dep ["myfile"] ["other.ml"]
  | _ -> ()
  end
```

1.2 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)`

1.3 ocamlfind

findlib

- *ocamlfind browser -all*
- *ocamlfind browser -package batteries*
- syntax extension
  ocamlfind ocamldep -package camlp4,xstrp4 -syntax camlp4r file1.ml file2.ml
  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 file (exmaple)

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

- simple Makefile for ocamlfind

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

1.4 toplevel

1. `#directory ‘‘_build’’ ;; #directory ‘‘+camlp4’’ ;; #load ‘‘...’’`

2. trace
3. labels (ignore labels in function types)
4. `warnings print_depth print_length`
5. hacking 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;;
     ```

     ```
     print_depth use principal untrace_all load list trace show directory u cd
     ```

     ```
     Topdirs.(dir_load,dir_use,dir_install_printer,dir_trace,dir_untrace,dir_
     ```

     ```
     - : (Format.formatter -> string -> unit) *
         (Format.formatter -> string -> unit) *
         (Format.formatter -> Longident.t -> unit) *
         (Format.formatter -> Longident.t -> unit) *
         (Format.formatter -> Longident.t -> unit) *
         (Format.formatter -> unit -> unit) *
         (Format.formatter -> string -> bool) * (unit -> unit) * (string -> un
     ```

   - store env

     ```
     let env = !Toploop.toplevel_env
     ... blabbla ...
     Toploop.toplevel_env := env
     ```

     ```
     Toploop.initialize_toplevel_env ()
     ```

   - **sample file for references** 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 nou
 * 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/li
  Topdirs.dir_load Format.err_formatter "/Users/bob/SourceCode/ML/godi/li
);;
```

- topfind.ml

  ideas : we can write **some utils** to check code later yeah. A poor man's code search tool (in the library `dir_top_level`)

```
se;;
- : ?ignore_module:bool -> (string -> bool) -> string -> string list =
se ~ignore_module:false (FILTER _*  "char" space* "->" space* "bool") "String";;
```

```
module Dont_use_this_name_ever :
    val contains : string -> char -> bool
    val contains_from : string -> int -> char -> bool
    val rcontains_from : string -> int -> char -> bool
    val filter : (char -> bool) -> string -> string
    module IString : sig type t = String.t val compare : t
       -> t -> int end
    module NumString : sig type t = String.t val compare : t
        -> t -> int end
```

```
module Exceptionless :
module Cap :
    val filter : (char -> bool) -> [> `Read ] t -> 'a t
    val contains : [> `Read ] t -> char -> bool
    val contains_from : [> `Read ] t -> int -> char ->
       bool
    val rcontains_from : [> `Read ] t -> int -> char ->
       bool
    module Exceptionless :
```

```
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 ())));;
#pwd;;
```

```
/Users/bob/SourceCode/Notes
```

1.5 git

- ignore set

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

Chapter 2

lexing

2.1 lexing-ulex-ocamllex

Ulex **support unicode**, while ocamllex don't, the tags file is as follows

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

Use default myocamlbuild.ml, like `ln -s ~/myocamlbuild.ml` and 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`

Ulex 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 |...
```

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.* Sometimes the semantics of rolling back is not what you want as recursive descent parser.

Abstraction with macro package

Since you need inline to do macro prepossessing, so use syntax extension macro to **inline** your code,

```
<*_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 files to `_build` if you use relative path in **INCLUDE** macro, otherwise you should use absolute path.

You can predefine some regexps (copied from ocaml source code) `parsing/lexer.ml`.

```
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'-'9']

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)
  (** here we have two quotient just to appeal the typesetting *)
| "\\" ['\\' '\'' '"' '"' '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
  (** for typesetting, duplication is not necessary *)
  | ['"' '"'] -> () (* end *)

  | '\\' newline ([' ' '\t'] * ) ->
        string lexbuf
  (** for typesetting, duplication is not necessary *)
  | '\\' ['\\' '\'' '"' '"' '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()
```

Ulex 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 (OCaml-type : `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` If you inspect the processed output by camlp4, you can see that the generated code *introducing Ulexing* very *late* and actually use very limited functions, other functions are just 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.

```
(** Runtime support for lexers generated by [ulex].
  This module is roughly equivalent to the module Lexing from
  the OCaml standard library, except that its lexbuffers handles
  Unicode code points (OCaml type: [int] in the range
  [0..0x10ffff]) instead of bytes (OCaml type: [char]).

  It is possible to have ulex-generated lexers work on a custom
  implementation for lex buffers. To do this, define a module [L] which
  implements the [start], [next], [mark] and [backtrack] functions
  (See the Internal Interface section below for a specification),
  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]

  Of course, you'll probably want to define functions like [lexeme]
  to be used in the lexers semantic actions.
*)

type lexbuf
  (** The type of lexer buffers. A lexer buffer is the argument passed
    to the scanning functions defined by the generated lexers.
    The lexer buffer holds the internal information for the
    scanners, including the code points of the token currently scanned,
    its position from the beginning of the input stream,
    and the current position of the lexer. *)

exception Error
  (** Raised by a lexer when it cannot parse a token from the lexbuf.
    The functions [Ulexing.lexeme_start] (resp. [Ulexing.lexeme_end]) can be
    used to find to positions of the first code point of the current
    matched substring (resp. the first code point that yield the error). *)

exception InvalidCodepoint of int
  (** Raised by some functions to signal that some code point is not
    compatible with a specified encoding. *)

(** {6 Clients interface} *)

val create: (int array -> int -> int -> int) -> lexbuf
  (** Create a generic lexer buffer.  When the lexer needs more
    characters, it will call the given function, giving it an array of
    integers [a], a position [pos] and a code point count [n].  The
    function should put [n] code points or less in [a], starting at
```

```
    position [pos], and return the number of characters provided. A
    return value of 0 means end of input. *)

val from_stream: int Stream.t -> lexbuf
  (** Create a lexbuf from a stream of Unicode code points. *)

val from_int_array: int array -> lexbuf
  (** Create a lexbuf from an array of Unicode code points. *)

val from_latin1_stream: char Stream.t -> lexbuf
  (** Create a lexbuf from a Latin1 encoded stream (ie a stream
    of Unicode code points in the range [0..255]) *)

val from_latin1_channel: in_channel -> lexbuf
  (** Create a lexbuf from a Latin1 encoded input channel.
    The client is responsible for closing the channel. *)

val from_latin1_string: string -> lexbuf
  (** Create a lexbuf from a Latin1 encoded string. *)

val from_utf8_stream: char Stream.t -> lexbuf
  (** Create a lexbuf from a UTF-8 encoded stream. *)

val from_utf8_channel: in_channel -> lexbuf
  (** Create a lexbuf from a UTF-8 encoded input channel. *)

val from_utf8_string: string -> lexbuf
  (** Create a lexbuf from a UTF-8 encoded string. *)

type enc = Ascii | Latin1 | Utf8
val from_var_enc_stream: enc ref -> char Stream.t -> lexbuf
```

```
  (** Create a lexbuf from a stream whose encoding is subject
    to change during lexing. The reference can be changed at any point.
    Note that bytes that have been consumed by the lexer buffer
    are not re-interpreted with the new encoding.

    In [Ascii] mode, non-ASCII bytes (ie [>127]) in the stream
    raise an [InvalidCodepoint] exception. *)

val from_var_enc_string: enc ref -> string -> lexbuf
  (** Same as [Ulexing.from_var_enc_stream] with a string as input. *)

val from_var_enc_channel: enc ref -> in_channel -> lexbuf
  (** Same as [Ulexing.from_var_enc_stream] with a channel as input. *)

(** {6 Interface for lexers semantic actions} *)

(** The following functions can be called from the semantic actions of
  lexer definitions.  They give access to the character string matched
  by the regular expression associated with the semantic action. These
  functions must be applied to the argument [lexbuf], which, in the
  code generated by [ulex], is bound to the lexer buffer passed to the
  parsing function.

  These functions can also be called when capturing a [Ulexing.Error]
  exception to retrieve the problematic string. *)

val lexeme_start: lexbuf -> int
  (** [Ulexing.lexeme_start lexbuf] returns the offset in the
    input stream of the first code point of the matched string.
    The first code point of the stream has offset 0. *)
```

```
val lexeme_end: lexbuf -> int
(** [Ulexing.lexeme_end lexbuf] returns the offset in the input stream
    of the character following the last code point of the matched
    string. The first character of the stream has offset 0. *)

val loc: lexbuf -> int * int
(** [Ulexing.loc lexbuf] returns the pair
   [(Ulexing.lexeme_start lexbuf,Ulexing.lexeme_end lexbuf)]. *)

val lexeme_length: lexbuf -> int
(** [Ulexing.loc lexbuf] returns the difference
   [(Ulexing.lexeme_end lexbuf) - (Ulexing.lexeme_start lexbuf)],
   that is, the length (in code points) of the matched string. *)

val lexeme: lexbuf -> int array
(** [Ulexing.lexeme lexbuf] returns the string matched by
   the regular expression as an array of Unicode code point. *)

val get_buf: lexbuf -> int array
  (** Direct access to the internal buffer. *)
val get_start: lexbuf -> int
  (** Direct access to the starting position of the lexeme in the
      internal buffer. *)
val get_pos: lexbuf -> int
  (** Direct access to the current position (end of lexeme) in the
      internal buffer. *)

val lexeme_char: lexbuf -> int -> int
  (** [Ulexing.lexeme_char lexbuf pos] returns code point number [pos] in
      the matched string. *)
```

```
val sub_lexeme: lexbuf -> int -> int -> int array
(** [Ulexing.lexeme lexbuf pos len] returns a substring of the string
  matched by the regular expression as an array of Unicode code point. *)


val latin1_lexeme: lexbuf -> string
(** As [Ulexing.lexeme] with a result encoded in Latin1.
  This function throws an exception [InvalidCodepoint] if it is not possible
  to encode the result in Latin1. *)

val latin1_sub_lexeme: lexbuf -> int -> int -> string
(** As [Ulexing.sub_lexeme] with a result encoded in Latin1.
  This function throws an exception [InvalidCodepoint] if it is not possible
  to encode the result in Latin1. *)

val latin1_lexeme_char: lexbuf -> int -> char
(** As [Ulexing.lexeme_char] with a result encoded in Latin1.
  This function throws an exception [InvalidCodepoint] if it is not possible
  to encode the result in Latin1. *)


val utf8_lexeme: lexbuf -> string
(** As [Ulexing.lexeme] with a result encoded in UTF-8. *)

val utf8_sub_lexeme: lexbuf -> int -> int -> string
(** As [Ulexing.sub_lexeme] with a result encoded in UTF-8. *)


val rollback: lexbuf -> unit
(** [Ulexing.rollback lexbuf] puts [lexbuf] back in its configuration before
  the last lexeme was matched. It is then possible to use another
```

```
  lexer to parse the same characters again. The other functions
  above in this section should not be used in the semantic action
  after a call to [Ulexing.rollback]. *)

(** {6 Internal interface} *)

(** These functions are used internally by the lexers. They could be used
  to write lexers by hand, or with a lexer generator different from
  [ulex]. The lexer buffers have a unique internal slot that can store
  an integer. They also store a "backtrack" position.
*)

val start: lexbuf -> unit
(** [Ulexing.start lexbuf] informs the lexer buffer that any
  code points until the current position can be discarded.
  The current position become the "start" position as returned
  by [Ulexing.lexeme_start]. Moreover, the internal slot is set to
  [-1] and the backtrack position is set to the current position.
*)

val next: lexbuf -> int
(** [Ulexing.next lexbuf next] extracts the next code point from the
  lexer buffer and increments to current position. If the input stream
  is exhausted, the function returns [-1]. *)

val mark: lexbuf -> int -> unit
(** [Ulexing.mark lexbuf i] stores the integer [i] in the internal
  slot. The backtrack position is set to the current position. *)

val backtrack: lexbuf -> int
(** [Ulexing.backtrack lexbuf] returns the value stored in the
```

```
  internal slot of the buffer, and performs backtracking
  (the current position is set to the value of the backtrack position). *)
```

Ulex does not handle line position, you have only global char position, but we are using emacs, not matter too much

ATTENTION

When you use ulex to generate the code, make sure to write the interface by yourself, the problem is that when you use the default interface, it will generate `__table__`, and different file may overlap this name, when you open the module, it will cause a disaster, so the best to do is write your `.mli` file.

And when you write lexer, make sure you write the default branch, check the generated code, otherwise its behavior is weird.

`camlp4of -parser macro pa_ulex.cma test_calc.ml -printer o` **Example**

Here is the example of simple basic lexer

```
open Ulexing
open Batteries

let regexp op_ar = ['+' '-' '*' '/']
let regexp op_bool = ['!' '&' '|']
let regexp rel = ['=' '<' '>']


(** get string output, not int array *)
let lexeme = Ulexing.utf8_lexeme

let rec basic = lexer
  | [' '] -> basic lexbuf
  | op_ar | op_bool ->
    let ar = lexeme lexbuf in
    `Lsymbol ar
  | "<=" | ">="| "<>" | rel ->
```

```
    `Lsymbol (lexeme lexbuf)
  |("REM" | "LET" | "PRINT"
       | "INPUT" | "IF"| "THEN") ->
    `Lsymbol (lexeme lexbuf)
  | '-'?['0'-'9']+ ->
    `Lint (int_of_string (lexeme lexbuf))
  | ['A'-'z']+ ->
    `Lident (lexeme lexbuf)
  | '"' [^ '"'] '"' ->
    `Lstring (let s = lexeme lexbuf in
              String.sub s 1 (String.length s - 2))
  | eof -> raise End_of_file
  | _ ->
    (print_endline (lexeme lexbuf ^ "unrecognized");
    basic lexbuf)

let token_of_string str =
  str
  |> Stream.of_string
  |> from_utf8_stream
  |> basic
let tokens_of_string str =
  let output = ref [] in
  let lexbuf = str |> Stream.of_string |> from_utf8_stream in
  (try
    while true do
    let token = basic lexbuf in
    output:= token :: !output;
    print_endline (dump token)
    done
  with End_of_file -> ());
```

```
  List.rev (!output)
```

```
let _ = tokens_of_string
  "a + b >= 3 > 3 < xx"
```

```
(**
assert_failure, assert_equal, @?, assert_raises, skip_if, todo, cmp_float
bracket
*)
let test_result = OUnit.(
  run_test_tt ("test-suite" >:::
                  ["test2" >:: (fun _ -> ());
                   "test1" >:: (fun _ -> "true" @? true)
                  ]
  ))
;;

(**Remark
*)
```

ocamllex

1. *module Lexing*

```
  se_str "from" "Lexing";;

val from_string : string -> lexbuf
val from_function : (string -> int -> int) -> lexbuf
val from_input : BatIO.input -> Lexing.lexbuf
val from_channel : BatIO.input -> Lexing.lexbuf
```

2. syntax

```
    {header}
    let ident = regexp ...
    rule entrypoint [arg1 .. argn ] =
       parse regexp {action }
       | ..
       | regexp {action}
    and entrypoint [arg1 .. argn] =
       parse ..
    and ...
    {trailer}
```

   The parse keyword can be replaced by shortest keyword.

   Typically, the header section contains the *open* directives required by the actions

   All identifiers starting with `__ocaml_lex` are reserved for use by **ocamllex**

3. example for me, best practice is put some test code in the trailer part, and use *ocamlbuild fc_lexer.byte* – to verify, or write a makefile. you can write several indifferent rule in a file using and.

```
(* verbatim translate *)
rule translate = parse
  | "current_directory" {print_string (Sys.getcwd ()); translate
      lexbuf}
  | _ as c {print_char c ; translate lexbuf}
  | eof {exit 0}

{
  let _ =
```

```
    let chan = open_in "fc_lexer.mll" in begin
    translate (Lexing.from_channel chan );
    close_in chan
    end

}
```

```
Legacy.Printexc.print;;
- : ('a -> 'b) -> 'a -> 'b = <fun>
```

4. caveat

   the longest(shortest) win, then consider the order of each regexp later. Actions are evaluated after the *lexbuf* is bound to the current lexer buffer and the identifier following the keyword *as* to the matched string.

5. position
   The lexing engine manages only the *pos_cnum* field of *lexbuf.lex_curr_p* with the number of chars read from the start of lexbuf. you are responsible for the other fields to be accurate. i.e.

```
let incr_linenum lexbuf = Lexing.(
    let pos = lexbuf.lex_curr_p in
    lexbuf.lex_curr_p <- { pos with
      pos_lnum = pos.pos_lnum + 1; (* line number *)
     pos_bol = pos.pos_cnum; (* the offset of the beginning of
        the
     line *)
    })
```

6. combine with ocamlyacc

   normally just add *open Parse* in the header, and use the token defined in *Parse*

7. tips

   (a) keyword table

```
   {let keyword_table = Hashtbl.create 72
    let _ = ...
  }
  rule token = parse
  | ['A'-'z' 'a'-'z'] ['A'-'z' 'A'-'z' '0'-'9' '_'] * as
     id
  {try Hashtbl.find keyword_table id with Not_found ->
     IDENT id}
  | ...
```

(b) for sharing **why ocamllex sucks**
some complex regexps are not easy to write, like string, but sharing is hard. To my knowledge, cpp preprocessor is fit for this task here. camlp4 is not fit, it will check other syntax, if you use ulex, camlp4 will do this job. So, my Makefile is part like this

```
lexer :
  cpp fc_lexer.mll.bak > fc_lexer.mll
  ocamlbuild -no-hygiene fc_lexer.byte --
```

even so, sharing is still very hard, since the built in compiler used another way to write string lexing. painful too sharing. so ulex wins in both aspects. sharing in ulex is much easier.

Chapter 3

parsing

3.1 ocamlyacc or menhir

We mainly cover menhir here.

A grammar is mainly composed of four elements(terminals, non-terminals, production rulls, start symbol)

Syntax

```
% {header
% }
%%
Grammar rules
%%
trailer
```

A tiny example as follows (It has a subtle bug, readers should find it)

```
% {
  open Printf
  let parse_error s =
    print_endline "error\n";
    print_endline s ;
    flush stdout
%}
```

```
%token <float> NUM
%token PLUS MINUS MULTIPLY DIVIDE CARET UMINUS
%token NEWLINE

%start input
%type <unit> input
%type <float> exp
%% /* rules and actions */

input: /* empty */ {}
    | input line {}
;

line: NEWLINE {}
    |exp NEWLINE  {printf "\t%.10g\n" $1 ; flush stdout}
;

exp: NUM { $1 }
    |exp exp PLUS {$1 +. $2 }
    |exp exp MINUS {$1 -. $2 }
    |exp exp MULTIPLY {$1 *. $2 }
    |exp exp DIVIDE {$1 /. $2 }
    |exp exp CARET {$1 ** $2 }
    |exp UMINUS {-. $1 }
;

%%
```

Notice that start non-terminal can be given *several*, then you will have a different .mli file, notice that it's different from ocamllex, ocamlyacc will generate a .mli file, so here we get the output interface as follows:

```
%type <type> nonterminal ... nonterminal
%start symbol ... symbol
```

```
type token =
  | NUM of (float)
  | PLUS
  | MINUS
  | MULTIPLY
  | DIVIDE
  | CARET
  | UMINUS
  | NEWLINE
val input :
  (Lexing.lexbuf  -> token) -> Lexing.lexbuf -> unit
val exp :
  (Lexing.lexbuf  -> token) -> Lexing.lexbuf -> float
```

Notice that we may use character strings as implicit terminals as in

```
expr : expr "+" expr {}
     | expr "*" expr {}
     | ... ;
```

They are directly processed by the parser without passing through the lexer. But it breaks the uniformity

Contextual Grammar

```
open Batteries

(**
   Grammar
   L := w C w
   w := (A|B)*
```

```
*)
type token = A |B |C

let rec parser1 = parser
  | [< 'A ; l = parser1 >] -> (parser [< 'A>] -> "a") :: l
  | [< 'B ; l = parser1 >] -> (parser [< 'B>] -> "b") :: l
  | [<>] -> []  (* always succeed *)
let parser2 lst str =
  List.fold_left (fun s p -> p str ^ s) ""  lst
let parser_L = parser
  | [< ls = parser1 ; 'C; r = parser2 ls >] ->
    r
let _ =
  [A;B;A;B;C;A;B;A;B]
  |>  Stream.of_list
  |>  parser_L
  |> print_endline
```

First gammar

```
/* empty corresponds Ctrl-d.*/
input : /*empty*/ {} | input line {};
```

Notice here we **preferred left-recursive** in yacc. The underlying theory for LALR prefers LR. because all the elements must be shifted onto the stack *before* the rule can be applied even once.

```
exp : NUM | exp exp PLUS | exp exp MINUS  ... ;
```

Here is our lexer

```
{
  open Rpcalc
  open Printf
```

```
  let first = ref true
}
let digit = ['0'-'9']
rule token = parse
  |[' ' '\t' ] {token lexbuf}
  |'\n' {NEWLINE}
  | (digit+ | "." digit+ | digit+ "." digit*) as num
      {NUM (float_of_string num)}
  |'+' {PLUS}
  |'-' {MINUS}
  |'*' {MULTIPLY}
  |'/' {DIVIDE}
  |'^' {CARET}
  |'n' {UMINUS}
  |_ as c  {printf "unrecognized char %c" c ; token lexbuf}
  |eof {
    if !first then begin first := false; NEWLINE end
    else raise End_of_file }

{
  let main ()  =
    let file = Sys.argv.(1) in
    let chan = open_in file in
    try
      let lexbuf = Lexing.from_channel chan in
      while true do
        Rpcalc.input token lexbuf
      done
    with End_of_file -> close_in chan

 let _ = Printexc.print main ()
```

```
}
```

We write driver function in lexer for convenience, since lexer depends on yacc. *Printex.print*

precedence associatitvity

Operator precedence is determined by the line ordering of the declarations;

%prec in the grammar section, the *%prec* simply instructs ocamlyacc that the rule *|Minus exp* has the same precedence as NEG *%left,%right,%nonassoc*

1. The associatitvity of an operator op determines how repeated uses of the operator nest: whether x *op* y *op* z is parsed by grouping x with y or. nonassoc will consider it as an error

2. All the tokens declared in a single precedence declaration have equal precedence and nest together according to their associatitvity

```
%{
  open Printf
  open Lexing
  let parse_error s =
    print_endline "impossible happend! panic \n";
    print_endline s ;
    flush stdout
%}

%token NEWLINE
%token LPAREN RPAREN
%token <float> NUM
%token PLUS MINUS MULTIPLY DIVIDE CARET


%left PLUS MINUS MULTIPLY DIVIDE NEG
```

```
%right CARET

%start input
%start exp
%type <unit> input
%type <float> exp

%% /* rules and actions */


input: /* empty */ {}
    | input line {}
;

line: NEWLINE {}
    |exp NEWLINE  {printf "\t%.10g\n" $1 ; flush stdout}
;

exp: NUM { $1 }
    | exp PLUS exp                { $1 +. $3 }
    | exp MINUS exp                { $1 -. $3 }
    | exp MULTIPLY exp                { $1 *. $3 }
    | exp DIVIDE exp                { $1 /. $3 }
    | MINUS exp %prec NEG        { -. $2 }
    | exp CARET exp                { $1 ** $3 }
    | LPAREN exp RPAREN                { $2 }
;

%%
```

Notice here the *NEG* is a place a holder, it takes the place, but it's not a token. since here we need *MINUS* has different levels. the interface file is as follows

Error Recovery

By default, the parser function raises exception after calling *parse_error* The ocamlyacc reserved word *error*

```
line: NEWLINE |exp NEWLINE | error NEWLINE {}
```

If an expression that cannot be evaluated is read, the error will be recognized by the third rule for line, and parsing will continue (parse_error is still called). This form of error recovery deals with syntax errors. There are also other kinds of errors.

Location Tracking

It's very easy. First, remember to use *Lexing.new_line* to track your line number, then use *rhs_start_pos, rhs_end_pos* to track the symbolposition. 1 is for the leftmost component.

```
Parsing.(
  let start_pos = rhs_start_pos 3 in
  let end_pos = rhs_end_pos 3 in
  printf "%d.%d --- %d.%d: dbz"
    start_pos.pos_lnum (start_pos.pos_cnum -start_pos.pos_bol)
    end_pos.pos_lnum (end_pos.pos_cnum - end_pos.pos_bol);
  1.0
)
```

For groupings, use the following function *symbol_start_pos, symbol_end_pos*

symbol_start_pos is set to the beginning of the leftmost component, and *symbol_end_pos* to the end of the rightmost component.

A complex Example

```
%{
  open Printf
  open Lexing
  let parse_error s =
    print_endline "impossible happend! panic \n";
```

```
    print_endline s ;
    flush stdout
  let var_table = Hashtbl.create 16
%}


%token NEWLINE
%token LPAREN RPAREN EQ
%token <float> NUM
%token PLUS MINUS MULTIPLY DIVIDE CARET
%token <string> VAR
%token <float->float>FNCT /* built in function */

%left PLUS MINUS
%left MULTIPLY DIVIDE
%left NEG

%right CARET
%start input
%start exp
%type <unit> input
%type <float> exp

%% /* rules and actions */


input: /* empty */ {}
    | input line {}
;

line: NEWLINE {}
```

```
    |exp NEWLINE  {printf "\t%.10g\n" $1 ; flush stdout}
    |error NEWLINE {}
;

exp: NUM { $1 }
    | VAR
        {try Hashtbl.find var_table $1
          with Not_found ->
            printf "unbound value '%s'\n" $1;
            0.0
        }
    | VAR EQ exp
        {Hashtbl.replace var_table $1 $3; $3}
    | FNCT LPAREN exp RPAREN
        { $1 $3 }
    | exp PLUS exp                { $1 +. $3 }
    | exp MINUS exp               { $1 -. $3 }
    | exp MULTIPLY exp            { $1 *. $3 }
    | exp DIVIDE exp
        { if $3 <> 0. then $1 /. $3
          else
            Parsing.(
              let start_pos = rhs_start_pos 3 in
              let end_pos = rhs_end_pos 3 in
              printf "%d.%d --- %d.%d: dbz"
                start_pos.pos_lnum (start_pos.pos_cnum -start_pos.pos_bol)
                end_pos.pos_lnum (end_pos.pos_cnum - end_pos.pos_bol);
              1.0
            )}
    | MINUS exp %prec NEG         { -. $2 }
    | exp CARET exp               { $1 ** $3 }
```

```
    | LPAREN exp RPAREN                { $2 }
;

%%



(** lexer file *)
{
  open Rpcalc
  open Printf
  let first = ref true
}


let digit = ['0'-'9']
let id = ['a'-'z']+
rule token = parse
  |[' ' '\t' ] {token lexbuf}
  |'\n' {Lexing.new_line lexbuf ; NEWLINE}
  | (digit+ | "." digit+ | digit+ "." digit*) as num
      {NUM (float_of_string num)}
  |'+' {PLUS}
  |'-' {MINUS}
  |'*' {MULTIPLY}
  |'/' {DIVIDE}
  |'^' {CARET}
  |'(' {LPAREN}
  |')' {RPAREN}
  |"sin" {FNCT(sin)}
  |"cos" {FNCT(cos) }
```

```
  |id as x {VAR x}
  |'=' {EQ}
  |_ as c  {printf "unrecognized char %c" c ; token lexbuf}
  |eof {
    if !first then begin first := false; NEWLINE end
    else raise End_of_file }


{
  let main ()  =
    let file = Sys.argv.(1) in
    let chan = open_in file in
    try
      let lexbuf = Lexing.from_channel chan in
      while true do
        Rpcalc.input token lexbuf
      done
    with End_of_file -> close_in chan

 let _ = Printexc.print main ()

}
```

In my opinion, the best practice is first modify .mly file, then change .mll file later

SHIFT REDUCE

A very nice tutorial shift-reduce

```
%token ID COMMA COLON
%token BOGUS /* NEVER LEX */
%start def
%type <unit>def
%%
```

```
def:    param_spec return_spec COMMA {}
        ;
param_spec:  ty {}
        |    name_list COLON ty {}
        ;

/*
return_spec:
             ty {}
        |    name COLON ty {}

        |    ID BOGUS {}   // This rule is never used
        ;
*/

/* another way to fix the prob */

return_spec : ty {}
        | ID COLON ty {}

ty:          ID {}
        ;
name:        ID {}
        ;
name_list:
             name {}
        |    name COMMA name_list {}
        ;



%{
```

```
%}


%token OPAREN CPAREN ID SEMIC DOT INT EQUAL

%start stmt
%type <int> stmt

%%
stmt: methodcall {0} | arrayasgn {0}
;

/*
previous
methodcall: target OPAREN CPAREN SEMIC {0}
;
target:  ID DOT ID {0} |ID {0}
;

our strategy was to remove the "extraneous" non-terminal in the
methodcall production, by moving one of the right-hand sides of target
to the methodcall production

*/

methodcall: target OPAREN CPAREN SEMIC {0} | ID OPAREN CPAREN SEMIC {0}
;
target:  ID DOT ID {0}
;
arrayasgn: ID OPAREN  INT CPAREN EQUAL INT SEMIC {0}
;
```

```
  %{
%}

%token RETURN ID SEMI EQ PLUS

%start methodbody
%type <unit> methodbody

%%

methodbody: stmtlist RETURN ID {}
;
/*
stmtlist: stmt stmtlist {} | stmt {}
;
the strategy here is simple, we use left-recursion instead of
right-recursion
*/

stmtlist: stmtlist stmt {} | stmt {}
;

stmt: RETURN ID SEMI {} | ID EQ ID PLUS ID {}
;

%{

%}
```

```
%token PLUS TIMES ID LPAREN RPAREN


%left PLUS
%left TIMES /* weird ocamlyacc can not detect typo TIMEs */

/*
here we add assiocaitivity and precedence
*/

%start expr
%type <unit> expr


%%

expr: expr PLUS expr {}
  | expr TIMES expr {}
  | ID {}
  | LPAREN expr RPAREN {}
;


%{

%}

%token ID EQ LPAREN RPAREN IF ELSE THEN


%nonassoc THEN
```

```
%nonassoc ELSE

/*
here we used a nice trick to
handle such ambiguity. set precedence of THEN, ELSE
both needed
*/

%start stmt
%type <unit> stmt

%%


stmt: ID EQ ID {}
  | IF LPAREN ID RPAREN THEN stmt {}
  | IF LPAREN ID RPAREN THEN stmt ELSE stmt {}



;
/*
It's tricky here we modify the grammar an unambiguous one
*/



/*
stmt        : matched {}
            | unmatched {}
            ;

matched     : IF '(' ID ')' matched ELSE matched {}
```

```
          ;

unmatched : IF '(' ID ')' matched {}
          | IF '(' ID ')' unmatched {}
          | IF '(' ID ')' matched ELSE unmatched {}
          ;
*/
%%
```

The prec trick is covered not correctly in this tutorial.

The symbols are declared to associate to the left, right, nonassoc. The symbols are *usually* tokens, they can also be *dummy* nonterminals, for use with the %prec directive in the rule.

1. Tokens and rules have precedences. The precedence of a *rule* is the precedence of its *rightmost* terminal. you can override this default by using the *%prec* directive in the rule

2. A reduce/reduce conflict is resolved in favor of the first ruel(in the order given by the source file)

3. A shift/reduce conflict is resolved by comparing the *predecence of the rule to be reduced* with the *precedence of the token to be shifted*. If the predecence of the rule is higher, then the rule will be reducecd; if the predecence of the token is higher then token will be shifted.

4. A shift/reduce conflict between a rule and a token with the same precedence will be resolved using the associativity.

5. when a shift/reduce can not be resolved, a warning, and in favor of *shift*

MENHIR Related

1. Syntax

```
specification ::= declaration . . . declaration %% rule . . .
   rule [ %% Objective Caml code ]
 declaration ::= %{ Objective Caml code %}
                         % parameter < uid : Objective Caml
                            module type >
                         %token [ < Objective Caml type > ] uid
                            . . . uid
                         %nonassoc uid . . . uid
                         %left uid . . . uid
                         %right uid . . . uid
                         %type < Objective Caml type > lid . . .
                             lid
                         %start [ < Objective Caml type > ] lid
                            . . . lid
rule ::= [%public] [%inline] lid [( id, ..., id)] : [|] group |
   ... | group
group ::= production | . . . | production { Objective Caml code
   } [ %prec id ]
production ::= producer . . . producer [ %prec id ]
 producer ::= [ lid = ] actual
actual ::= id [( actual, ..., actual)] [? | + | *]

\item parameter

  \begin{bluetext}
%parameter <uid: Objective module types>
```

This causes the entire parser to be parameterized over the module.

2. multiple files (private and public,tokens aside)

3. parameterized rules

4. inline

5. standard library

Name	Recognizes	Produces	Comment
option(X)	ϵ | X	α option, if X : α	
ioption(X)			inlined
boption(X)			bool
loption(X)	ϵ |X	α list, if X : α list	
pair(X,Y)	X Y	$\alpha \times \beta$	
separated_pair(X,sep,Y)	X sep Y	$\alpha \times \beta$	
preceded(opening,X)	opening X	α, if X : α	
terminated(X,closing)	X closing	α, if X : α	
delimited(opening, X closing)	opening X closing	α, if X : α	
list(X)			
nonempty_list(X)			
separated_list(sep,X)			
sepearted_nonempty_list(sep,X)			

6. combined with ulex

   A typical tags file is as follows

```
true:use_menhir, pkg_ulex, pkg_pcre, pkg_menhirLib,
   pkg_batteries
<scanner.ml>: pkg_ulex, syntax_camlp4o
```

   You have to use

```
Menhirlib.Convert
```

   API, here

```
(** support  ocamllex *)
type ('token, 'semantic_value) traditional =
    (Lexing.lexbuf -> 'token) -> Lexing.lexbuf -> 'semantic_value


(**
  This revised API is independent of any lexer generator. Here, the
```

```
   parser only requires access to the lexer, and the lexer takes no
   parameters. The tokens returned by the lexer *may*  need to contain
   position  information. *)

type ('token, 'semantic_value) revised =
    (unit -> 'token) -> 'semantic_value

(* A token of the revised lexer is essentially a triple of a token
   of the traditional lexer (or raw token), a start position, and
   and end position. The three [get] functions are accessors. *)

(* We do not require the type ['token] to actually be a triple type.
   This enables complex applications where it is a record type with
   more than three fields. It also enables simple applications where
   positions are of no interest, so ['token] is just ['raw_token]
   and [get_startp] and [get_endp] return dummy positions. *)

val traditional2revised:
  ('token -> 'raw_token) ->
  ('token -> Lexing.position) ->  (* get a a start position *)
  ('token -> Lexing.position) ->  (* get an end position *)
  ('raw_token, 'semantic_value) traditional ->
  ('token, 'semantic_value) revised

val revised2traditional:
  ('raw_token -> Lexing.position -> Lexing.position -> 'token) ->
  ('token, 'semantic_value) revised ->
  ('raw_token, 'semantic_value) traditional

(** concrete type used here *)
module Simplified : sig
```

```
  val traditional2revised:
    ('token, 'semantic_value) traditional ->
    ('token * Lexing.position * Lexing.position, 'semantic_value) revised
  val revised2traditional:
    ('token * Lexing.position * Lexing.position, 'semantic_value) revised ->
    ('token, 'semantic_value) traditional
end
```

7. example csss project

Chapter 4

Camlp4

Camlp4 stands for preprocess-pretty-printer for `OCaml`, it's extremely powerful and hard to grasp as well.

4.1 Breif intro to parser

A brief intro to recursive descent parser.

Grammar transform

```
a : a x | b (x can be anything)
=>
a : b r
r : x r | e
-----------------------
exp : exp op exp | prim
=>
exp : prim expR
expR : op exp expR | e
```

4.2 Basics Command Lines

```
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
```

You can grep all executables relevant to camlp4 using a one-line bash as follows:

```
find $(dirname $(which ocaml)) -type f -perm -og+rx | grep camlp4 |
while read ss ; do echo $(basename $ss) ; done
```

```
camlp4
camlp4boot
camlp4o
camlp4o.opt
camlp4of
camlp4of.opt
camlp4oof
camlp4oof.opt
camlp4orf
camlp4orf.opt
camlp4prof
camlp4r
camlp4r.opt
camlp4rf
camlp4rf.opt
mkcamlp4
safe_camlp4
```

So the tools at hand are **camlp4, camlp4o, camlp4of, camlp4oof, camlp4orf, camlp4r, camlp4rf**

```
camlp4 -h

Usage: camlp4 [load-options] [--] [other-options]
Options:
<file>.ml        Parse this implementation file
<file>.mli       Parse this interface file
<file>.(cmo|cma) Load this module inside the Camlp4 core
  -I <directory>   Add directory in search patch for object files.
  -where           Print camlp4 library directory and exit.
  -nolib           No automatic search for object files in library
     directory.
  -intf <file>     Parse <file> as an interface, whatever its
     extension.
  -impl <file>     Parse <file> as an implementation, whatever its
     extension.
```

```
-str <string>     Parse <string> as an implementation.
-unsafe           Generate unsafe accesses to array and strings.
-noassert         Obsolete, do not use this option.
-verbose          More verbose in parsing errors.
-loc <name>       Name of the location variable (default: _loc).
-QD <file>        Dump quotation expander result in case of syntax
   error.
-o <file>         Output on <file> instead of standard output.
-v                Print Camlp4 version and exit.
-version          Print Camlp4 version number and exit.
-vnum             Print Camlp4 version number and exit.
-no_quot          Don't parse quotations, allowing to use, e.g.
   "<:>" as token.
-loaded-modules   Print the list of loaded modules.
-parser <name>    Load the parser Camlp4Parsers/<name>.cm(o|a|xs)
-printer <name>   Load the printer Camlp4Printers/<name>.cm(o|a|xs)
-filter <name>    Load the filter Camlp4Filters/<name>.cm(o|a|xs)
-ignore           ignore the next argument
--                Deprecated, does nothing
```

Useful options

`-str`

`-loaded-modules`

`-parser <name>` load the parser *Camlp4Parsers/<name>.cm(o|a|xs)*

`-printer <name>` load the printer *Camlp4Printerss/<name>.cm(o|a|xs)*

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

`-printer o` means print in original syntax.

These command lineoptions are all handled in /Camlp4Bin.ml |

`Camlp4o -h` There are options added by loaded object files

`-add_locations` Add locations as comment

`-no_comments`

`-curry-constr`

`-sep` Use this string between parsers

That reflective is true means when extending the syntax of the host language will **also extend the embedded 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

Camlp4r

1. parser
   RP, RPP(RevisedParserParser)

2. printer
   OCaml

Camlp4rf (extended from camlp4r)

1. parser
   RP,RPP, GrammarP, ListComprehension, MacroP, QuotationExpander

2. printer
   OCaml

Camlp4o (extended from camlp4r)

1. parser
   OP, OPP, RP,RPP

Camlp4of (extended from camlp4o)

1. parser
   GrammarParser, ListComprehension, MacroP, QuotatuinExpander

2. printer

Without ocamlbuild, ocamlfind, a simple build would be like this

```
ocamlc -pp camlp4o.opt error.ml
```

```
camlp4of -str "let a = [x| x <- [1.. 10] ] "
let a = [ 1..10 ]
camlp4o -str 'true && false'
true && false
```

```
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))
```

Now we begin to explore the structure of camlp4 Source Code

First let's have a look at the directory structure of camlp4 directory.

```
|<.>
|--<boot>
|--<build>
|--<Camlp4>
|----<Printers>
|----<Struct>        -- important
|------<Grammar>
|--<Camlp4Filters>   -- important
|--<Camlp4Parsers>   -- important
|--<Camlp4Printers>
|--<Camlp4Top>
|--<examples>        -- important
|--<man>
|--<test>
|----<fixtures>
|--<unmaintained>    -- many useful extensions unmatained
```

```
|----<compile>
|----<etc>
|----<extfold>        -- fold extension
|----<format>
|----<lefteval>
|----<lib>
|----<ocamllex>
|----<ocpp>
|----<odyl>
|----<olabl>
|----<scheme>
|----<sml>
```

`Camlp4.PreCast (Camlp4/PreCast.ml)`

Struct directory has module *Loc, Dynloader Functor, Camlp4Ast.Make, Token.Make, Lexer.Make, Grammar.Static.Make, Quotation.Make*

Camlp4.PreCast **re-export** such files

```
    Struct/Loc.ml
    Struct/Camlp4Ast.mlast
    Struct/Token.ml
    Struct/Grammar/Parser.ml
    Struct/Grammar/Static.ml
    Struct/Lexer.mll
    Struct/DynLoader.ml
    Struct/Quotation.ml
    Struct/AstFilters.ml
    OCamlInitSyntax.ml
    Printers/OCaml.ml
    Printers/OCamlr.ml
    Printers/Null.ml
    Printers/DumpCamlp4Ast.ml
    Printers/DumpOCamlAst.ml
```

```
(** Camlp4.PreCast.ml *)
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                  (* interesting *)
  | LIDENT        of string
  | UIDENT        of string
  | ESCAPED_IDENT of string                  (* interesting *)
  | 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                  (* interesting *)
  | BLANKS        of string                  (* interesting *)
  | NEWLINE                                  (* interesting *)
  | LINE_DIRECTIVE of int and option string  (* interesting *)
  | EOI ];


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;

(** intersting, so you can make your own syntax totally
```

```
    but it's not easy to do this in toplevel, probably will crash..
    *)
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;


(** Camlp4.OCamlInitSyntax.ml
    Ast -> Gram -> Quotation -> Camlp4Syntax
    Given Ast, Gram, Quotation, we produce Camlp4Syntax
 *)
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
 module Quotation = Quotation ;
 value parse_implem ...
```

```
 value parse_interf ...
 value print_interf ...
 value print_implem ...
 module Quotation = Quotation ;
end
```

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

`Camlp4.Sig.ml` All are signatures, there's even no `Camlp4.Sig.mli`

`Camlp4.Struct.Camlp4Ast.mlast` This file use macro `INCLUDE` to include `Camlp4.Camlp4Ast.pa`
for reuse.

```
(** file Camlp4Ast.mlast
  in the file we have *)
Camlp4.Struct.Camlp4Ast.Make : Loc -> Sig.Camlp4Syntax
  module Ast = struct
     include Sig.MakeCamlp4Ast Loc
  end ;
```

Another utility, you can inspect what modules you have loaded in toplevel:

```
Camlp4.Register.loaded_modules;;
- : string list ref =
{Pervasives.contents =
  ["Camlp4GrammarParser"; "Camlp4OCamlParserParser";
   "Camlp4OCamlRevisedParserParser"; "Camlp4OCamlParser";
   "Camlp4OCamlRevisedParser"]}
```

4.3 Ast Transformation

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*.

Camlp4 uses the **filter** iteself to bootstrap.

```
(** file Camlp4Ast.mlast *)
class map = Camlp4MapGenerator.generated;
class fold = Camlp4FoldGenerator.generated;
```

As above, `Camlp4.Ast` has a corresponding map traversal object, which could be used by you: (the class was generated by our filter) `Ast.map` is a class

```
let b = new Camlp4.PreCast.Ast.map ;;
val b : Camlp4.PreCast.Ast.map = <obj>
```

```
(** a simple ast transform *)
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 AstFilters.register_str_item_filter simplify#str_item
```

you can write it without sytax extension(very tedious),

```
(** the same as above without syntax extension, you can get with
    camlp4of ast_add_zero.ml -printer o *)
let _ =
  let simplify =
object
  inherit Ast.map as super
  method expr =
    fun e ->
      match super#expr e with
        | Ast.ExApp (_,
                     (Ast.ExApp (_, (Ast.ExId (_, (Ast.IdLid (_, "+")))), x)),
```

```
                    (Ast.ExInt (_, "0"))) |
              Ast.ExApp (_,
                    (Ast.ExApp (_, (Ast.ExId (_, (Ast.IdLid (_, "+")))),
                              (Ast.ExInt (_, "0")))),
                    x)
          -> x
        | x -> x
end
  in AstFilters.register_str_item_filter simplify#str_item
```

To make life easier, you can write like this

```
let _ =
  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
```

In the module `Camlp4.PreCast.AstFilters`, there are some utiliies to do filter over the ast.

```
type 'a filter = 'a -> 'a
val register_sig_item_filter : Ast.sig_item filter -> unit
val register_str_item_filter : Ast.str_item filter -> unit
val register_topphrase_filter : Ast.str_item filter -> unit
val fold_interf_filters : ('a -> Ast.sig_item filter -> 'a) -> 'a -> 'a
val fold_implem_filters : ('a -> Ast.str_item filter -> 'a) -> 'a -> 'a
val fold_topphrase_filters :
  ('a -> Ast.str_item filter -> 'a) -> 'a -> 'a
```

You can also generate map traversal for ocaml type. *put your type definition before* you macro, like this

```
type a =
  | A of b
  | C
and b =
  | B of a
  | D

class map = Camlp4MapGenerator.generated

let _ =
  let v = object
    inherit map as super
    method! b x = match super#b x with
      | D -> B C
      | x -> x
  end in
  assert (v#b D = B (C))
```

Without filter, you would write the transformer by hand like this

```
(** The processed output of ast_map *)
type a = | A of b | C and b = | B of a | D

class map =
  object ((o : 'self_type))
    method b : b -> b = function | B _x -> let _x = o#a _x in B _x | D -> D
```

```
    method a : a -> a = function | A _x -> let _x = o#b _x in A _x | C -> C
    method unknown : 'a. 'a -> 'a = fun x -> x
  end

let _ =
  let v =
    object
      inherit map as super
      method! b = fun x -> match super#b x with | D -> B C | x -> x
    end
  in assert ((v#b D) = (B C))
```

Camlp4 use the filter in `antiquot_expander`, for example in *Camlp4Parsers/Camlp4QuotationC* in the definition of `add_quotation`, we have

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

Notice that it first invoked `parse_quot_string`, then do some transformation, **that's how quotation works** !, it will be changed to your customized quotation parser, and when it goes to antiquot syntax, it will go back to **host language parser**. Since the host language parser also support quotation syntax (due to **reflexivity**), so you **nest your quotation whatever you want.**

4.4 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 (*camlp4o -str "type t = ('a,bool) foo" -printer r  -> type t = 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}


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 ;


   [<'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
```

4.5 Experimentation Environment

On Toplevel **via findlib**

```
ocaml
#camlp4r;
#load "camlp4rf.cma"
```

Using ocamlobjinfo to search modules:

```
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
```

Using **script** (oco using original syntax is ok), but when using ocr(default revised syntax), it will have some problems, i.e. .ocamlinit, and other startup files including findlib, so you'd better not use revised syntax in the toplevel. here I use .ocamlinitr (revised syntax) for ocr, but it still have some problem with findlib, (internal, hard to solve), but it does not really matter.

```
bash-3.2$ cat /usr/local/bin/oco
ledit -x -h ~/.ocaml_history ocaml dynlink.cma camlp4of.cma -warn-error +a-4-6-27..29
cat `which ocr`
ledit -x -h ~/.ocaml_history ocaml dynlink.cma camlp4rf.cma -init ~/.ocamlinitr -warn-error +a-4-6-27..29
```

4.6 Extensible Parser

Camlp4's extensible parser is deeply combined with its own lexer, use menhir if it is very complex and not ocaml-oriented. It is very hard to debug in itself. So I suggest it is used to do simple ocaml-oriented parsing.

4.6.1 Examples

First example (a simple calculation)

```
open Camlp4.PreCast

let expression = Gram.Entry.mk "expression"

let _ =
  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

let  _ =
  Printf.printf "%d" (
    Gram.parse_string
      expression
```

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

The tags file is

```
<simple_calc.ml> : pp(camlp4of)
<simple_calc.{cmo,byte,native}> : use_dynlink, use_camlp4_full
```

For oco in **toplevel** , extensible parser works **quite well in original syntax**, so if you don't do quasiquoation in toplevel, *feel free to use original syntax.*

Some keywords for extensible paser

```
EXTEND END  LIST0 LIST1 SEP TRY SELF OPT  FIRST LAST  LEVEL AFTER BEFORE
```

SELF represents either the **current level**, **the next level** or the **first level** depending on the **associativity** and the **position** of the SELF in the rule .

The identifier NEXT, which is a call to the next level of the current entry.

4.6.2 Mechanism

A brief introduction to its mechanism

There are four generally four phases

1 collection of new keywords, and update of the lexer associated to the grammar

2 representation of the grammar as a tree data structure

3 left-factoring of each precedence level
when there's a common perfix of symblos(a symbol is a keyword, token, or entry), the parser does not branch until the common parser has been parsed. **that's how grammars are implemented, first the corresponding tree is generated, then the parser is generated for the tree.** some tiny bits

  (i) Greedy first
  when one rule is a prefix of another. **a token or keyword is preferred over epsilon, the empty string (this also holds for other ways that a**

grammar can match epsilon) factoring happens when the parser is built .

(ii) **explicit token or keyword trumps an entry** so you have two prductions, with the same prefix, except the last one. one is another entry, and the other is a token, **the parser will first try the token, if it succeeds, it stops, otherwise they try the entry.** This sounds weird, but it is reasonable, after left-factorization, the parser pays no cost when it tries just a token, it's amazing that even more tokens, the token rule still wins, and **even the token rule fails after consuming some tokens, it can even transfer to the entry rule** , local try????? . **it seems that after factorization, the rule order may be changed** .

(iii) the data structure representing the grammar is then passed as argument to a generic parser

It's really hard to understand how it really works. Here are some experiments I did, but did not know how to explain

```
(** #require "camlp4.gramlib"*)
open Camlp4.PreCast

module MGram = MakeGram(Lexer)

let m_expr = MGram.Entry.mk "m_expr";;
let pr = MGram.Entry.print
let _ =
  EXTEND MGram
    GLOBAL: m_expr ;
    m_expr :
      [[ "foo"; f  -> print_endline "first"
       | "foo" ; "bar"; "baz" -> print_endline "second"]
      ];
```

```
    f : [["bar"; "baz" ]];  END;;

let _ = pr Format.std_formatter m_expr
let _ = MGram.parse_string m_expr (Loc.mk "<string>") "foo bar baz ";;

(** output
    m_expr: [ LEFTA
    [ "foo"; "bar"; "baz"
    | "foo"; f ] ]
    second
*)


(** DELETE_RULE expr: SELF; "+"; SELF END;; *)
let _ = begin
  MGram.Entry.clear m_expr;
  EXTEND MGram GLOBAL: m_expr ;
    m_expr :
     [[ "foo"; f  -> print_endline "first"
      | "foo" ; "bar"; "bax" -> print_endline "second"]
     ];
    f : [["bar"; "baz" ]];
  END;
  pr Format.std_formatter m_expr ;
  MGram.parse_string m_expr (Loc.mk "<string>") "foo bar baz "
end
(** output:
   m_expr: [ LEFTA
    [ "foo"; "bar"; "bax"
    | "foo"; f ] ]
  first
```

```
*)

let _ = begin
  MGram.Entry.clear m_expr;
  EXTEND MGram GLOBAL: m_expr ;
    m_expr :
     [[ "foo"; f  -> print_endline "first"
      | "foo" ; "bar"; f -> print_endline "second"]
     ];
    f : [["bar"; "baz" ]];
  END;
  pr Format.std_formatter m_expr;
  MGram.parse_string m_expr (Loc.mk "<string>") "foo bar baz "
end
(**
   m_expr: [ LEFTA
   [ "foo"; "bar"; f
   | "foo"; f ] ]
   Exception:
   Loc.Exc_located (<abstr>,
   Stream.Error "[f] expected after \"bar\" (in [m_expr])").
*)
```

We see that `MGram.Entry.print` is a good utility.

The processed code is not too indicative, all the dispatch magic hides in `MGram.extend` function (or |Insert.extend| function) *camlp4/Camlp4/Struct/Grammar/Insert.ml*

```
value extend entry (position, rules) =
      let elev = levels_of_rules entry position rules in
      do {
        entry.edesc := Dlevels elev;
        entry.estart :=
```

```
          fun lev strm ->
            let f = Parser.start_parser_of_entry entry in
            do { entry.estart := f; f lev strm };
        entry.econtinue :=
          fun lev bp a strm ->
            let f = Parser.continue_parser_of_entry entry in
            do { entry.econtinue := f; f lev bp a strm }
      };
```

Factoring only happens in the same level within a rule.

You can do explicit backtracking by hand (npeek trick)

```
(**hand-coded entry  MGram.Entry.of_parser *)
open Camlp4.PreCast
module MGram = MakeGram(Lexer)
let pr = MGram.Entry.print
let test = MGram.Entry.of_parser "test"
  (fun strm -> match Stream.npeek 2 strm with
      [_ ; KEYWORD "xyzzy", _ ] -> raise Stream.Failure
    | _ -> ())
let m_expr = MGram.Entry.mk "m_expr"

let _ = begin
  EXTEND MGram GLOBAL: m_expr ;
  g : [[ "plugh" ]] ;
  f1 : [[ g ; "quux" ]];
  f2 : [[g ; "xyzzy"]] ;
  m_expr : [[test ; f1 -> print_endline "1" | f2 -> print_endline "2" ]] ;
  END ;
  pr Format.std_formatter m_expr;
  MGram.parse_string m_expr (Loc.mk "<string>") "plugh xyzzy"
end
```

```
(**
  m_expr: [ LEFTA
  [ test; f1
  | f2 ] ]
  2
*)
```

(a) left factorization

take rules as follows as an example

```
"method"; "private"; "virtual"; l = label; ":"; t = poly_type
"method"; "virtual"; "private"; l = label; ":"; t = poly_type
"method"; "virtual"; l = label; ":"; t = poly_type
"method"; "private"; l = label; ":"; t = poly_type; "="; e = expr
"method"; "private"; l = label; sb = fun_binding
"method"; l = label; ":"; t = poly_type; "="; e = expr
"method"; l = label; sb = fun_binding
```

The rules are inserted in a tree and the result looks like:

```
"method"
   |-- "private"
   |       |-- "virtual"
   |       |       |-- label
   |       |             |-- ":"
   |       |                  |-- poly_type
   |       |-- label
   |             |-- ":"
   |             |    |-- poly_type
   |             |            |-- ":="
   |             |                 |-- expr
   |             |-- fun_binding
```

```
|-- "virtual"
|       |-- "private"
|       |       |-- label
|       |             |-- ":"
|       |                  |-- poly_type
|       |-- label
|             |-- ":"
|                  |-- poly_type
|-- label
      |-- ":"
      |     |-- poly_type
      |             |-- "="
      |                  |-- expr
      |-- fun_binding
```

This tree is built as long as rules are inserted.

(b) **start and continue** At each entry level, the rules are separated into **two trees**:

   (a) The tree of the rules not starting with neither the current entry name nor by "SELF"(start)

   (b) The tree of the rules starting with the current entry or by SELF, this symbol **itself not being included** in the tree

   They determine two functions :

   (a) The function named "start", analyzing the first tree

   (b) The function named "continue", taking, as parameter, a value previously parsed, and analyzing the second tree.

   A call to an entry, correspond to a call to the **"start"** function of the **"first"** level of the entry.

   For the "start", it tries its tree, if it works, it calls the "continue" function of the same level, giving the result of "start" as parameter. If this "continue" fails, return itself. (continue may do some more interesting stuff). If the "start" function fails, the "start" of the next level is tested until it fails.

For the "continue", it first tries the "continue" function of the **next** level. (here + give into *), if it fails or it's the last level, it then tries itself, giving the result as parameter. If it still fails, return its extra parameter.

A special case for rules ending with SELF or the current entry name. For this last symbol, there's a call to the "start" function of **the current level (RIGHTA) or the next level (OTHERWISE)**

When a SELF or the current entry name is encountered in the middle of the rule, there's a call to the start of the **first level** of the current entry.

Each entry has a start and continue

```
(* 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 (* backtracking *)
FIRST LAST LEVEL level, AFTER level, BEFORE level
```

4.6.3 STREAM PARSER

(a) stream parser

```
let rec p = parser [< '"foo"; 'x ; '"bar">] -> x | [< '"baz"; y = p >] -> y;;
val p : string Batteries.Stream.t -> string = <fun>
```

```
camlp4of  -str "let rec p = parser [< '\"foo\"; 'x ; '\"bar\">] -> x | [< '\"ba
(** output
   normal pattern : first peek, then junk it
*)
let rec p (__strm : _ Stream.t) =
  match Stream.peek __strm with
  | Some "foo" ->
      (Stream.junk __strm;
       (match Stream.peek __strm with
        | Some x ->
            (Stream.junk __strm;
             (match Stream.peek __strm with
              | Some "bar" -> (Stream.junk __strm; x)
              | _ -> raise (Stream.Error "")))
        | _ -> raise (Stream.Error "")))
  | Some "baz" ->
      (Stream.junk __strm;
       (try p __strm with | Stream.Failure -> raise (Stream.Error "")))
  | _ -> raise Stream.Failure
camlp4of -str "let rec p = parser [< x = q >] -> x | [< '\"bar\">] -> \"bar\""
(** output  *)
let rec p (__strm : _ Stream.t) =
  try q __strm
  with
  | Stream.Failure -> (* limited backtracking *)
      (match Stream.peek __strm with
       | Some "bar" -> (Stream.junk __strm; "bar")
       | _ -> raise Stream.Failure)
```

4.6.4 Grammar

```
se (FILTER _* "Exc_located") "Loc" ;;
exception Exc_located of t * exn
(** an exception containing an exception *)
se (FILTER _* "type" space+ "t") "Loc";;
type t = Camlp4.PreCast.Loc.t
```

we can re-raise the exception so it gets *printed* using *Printexc* .

A literal string (like "foo") indicates a **KEYWORD** token ; using it in a grammar **registers the keyword** with the lexer. When it is promoted as a key word, it will no longer be used as a **LIDENT**, so for example, the parser parser, will **break some valid programs** before, because **parser** is now a keyword. This is the convention, to make things simple, you can find other ways to overcome the problem, but it's too complicated. you can also say (x= KEYWORD) or pattern match syntax ('LINDENT x) to get the actual token constructor. The parser **ignores** extra tokens after a success.

1. LEVELS
   they can be labeled following an entry, like (expr LEVEL "mul"). However, explicitly specifying a level when calling an entry `might defeats` the start/-continue mechanism.

2. NEXT LIST0 SEP OPT TRY
   NEXT refers to the entry being defined at the following level regardless of assocaitivity or position. LIST0 elem SEP sep . Both LIST0 and OPT can match the epsilon, but its priority is lower. For TRY, non-local backtracking, a Stream.Error will be converted to a Stream.Failure.

   ```
   expr : [[ TRY f1 -> "f1" | f2 -> "f2" ]]
   ```

3. nested rule (only one level)

   ```
   [x = expr ; ["+" | "plus" ]; y = expr -> x + y ]
   ```

4. EXTEND is an expression (of type unit)
   it can be evaluated at toplevel, but also inside a function, when the syntax extension takes place when the function is called.

5. Translated sample code

```
open Camlp4.PreCast
module MGram = MakeGram(Lexer)
EXTEND MGram
   GLOBAL: m_expr ;
    m_expr :
     [[ "foo"; f  -> print_endline "first"
      | "foo" ; "bar"; "bax" -> print_endline "second"]
     ];
    f : [["bar"; "baz" ]];  END;;
```

The processed code is as follows:

```
(** translated code output *)
open Camlp4.PreCast
module MGram = MakeGram(Lexer)
let _ =
  let _ = (m_expr : 'm_expr MGram.Entry.t) in
  let grammar_entry_create = MGram.Entry.mk in
  let f : 'f MGram.Entry.t = grammar_entry_create "f"
  in
    (MGram.extend (m_expr : 'm_expr MGram.Entry.t)
       ((fun () ->
           (None,
            [ (None, None,
               [ ([ MGram.Skeyword "foo"; MGram.Skeyword "bar";
```

```
                      MGram.Skeyword "bax" ],
                     (MGram.Action.mk
                        (fun _ _ _ (_loc : MGram.Loc.t) ->
                           (print_endline "second" : 'm_expr))));
                    ([ MGram.Skeyword "foo";
                      MGram.Snterm (MGram.Entry.obj (f : 'f MGram.Entry.t)) ],
                     (MGram.Action.mk
                        (fun _ _ (_loc : MGram.Loc.t) ->
                           (print_endline "first" : 'm_expr)))) ]) ]))
              ());
         MGram.extend (f : 'f MGram.Entry.t)
           ((fun () ->
               (None,
                [ (None, None,
                   [ ([ MGram.Skeyword "bar"; MGram.Skeyword "baz" ],
                     (MGram.Action.mk
                        (fun _ _ (_loc : MGram.Loc.t) -> (() : 'f)))) ]) ]))
              ()))
```

6. if there are unexpected symbols after a correct expression, the trailing symbols are ignored.

   ```
   let expr_eoi = Grammar.Entry.mk "expr_eoi" ;;
   EXTEND expr_eoi : [[ e = expr ; EOI -> e]]; END ;;
   ```

   The keywords are stored **in a hashtbl**, so it can be updated `dynamically`.

7. level

   ```
   rule ::= list-of-symbols-seperated-by-semicolons -> action
   level ::=  optional-label optional-associativity
   [list-of-rules-operated-by-bars]
   entry-extension ::=
   ```

```
identifier : optional-position  [ list-of-levels-seperated-by-
    bars ]
optional-position ::= FIRST | LAST | BEFORE label | AFTER
    label |
LEVEL label
```

8. Grammar modification

   When you extend an entry, by default **the first level of the extension extends the first level of the entry**

   For example you a grammar like this :

```
(* #require "camlp4.gramlib";; *)

open Camlp4.PreCast
module MGram = MakeGram(Lexer)

let test = MGram.Entry.mk "test"
let p = MGram.Entry.print


let _ = begin
  MGram.Entry.clear test;
  EXTEND MGram GLOBAL: test;
  test:
    ["add" LEFTA
        [SELF; "+" ; SELF | SELF; "-" ; SELF]
    | "mult" RIGHTA
        [SELF; "*" ; SELF | SELF; "/" ; SELF]
    | "simple" NONA
        [ "("; SELF; ")"  | INT ] ];
  END;
  p Format.std_formatter test;
end
```

```
(** output
 test: [ "add" LEFTA
        [ SELF; "+"; SELF
        | SELF; "-"; SELF ]
      | "mult" RIGHTA
        [ SELF; "*"; SELF
        | SELF; "/"; SELF ]
      | "simple" NONA
        [ "("; SELF; ")"
        | INT ((_)) ] ]

*)



let _ = begin
  EXTEND MGram GLOBAL: test;
    test: [[ x = SELF ; "plus1plus" ; y = SELF ]];
  END ;
  p Format.std_formatter test
end
(** output
test: [ "add" LEFTA
    [ SELF; "plus1plus"; SELF
  | SELF; "+"; SELF
  | SELF; "-"; SELF ]
| "mult" RIGHTA
  [ SELF; "*"; SELF
  | SELF; "/"; SELF ]
| "simple" NONA
```

```
  [ "("; SELF; ")"
  | INT ((_)) ] ]
*)
```

This extends the first level “add”. you can double check by printing the result

When you want to create a new level in the last position

```
let _ = begin
  EXTEND MGram  test: LAST
    [[x = SELF ; "plus1plus" ; y = SELF ]];
  END;
  p Format.std_formatter test
end
(** output
              test: [ "add" LEFTA
  [ SELF; "plus1plus"; SELF
  | SELF; "+"; SELF
  | SELF; "-"; SELF ]
| "mult" RIGHTA
  [ SELF; "*"; SELF
  | SELF; "/"; SELF ]
| "simple" NONA
  [ "("; SELF; ")"
  | INT ((_)) ]
| LEFTA
  [ SELF; "plus1plus"; SELF ] ]
*)
```

When you want to insert in the level “mult” in the first position

```
let _ = begin
  EXTEND MGram  test: LEVEL "mult" [[x = SELF ; "plus1plus" ; y = SELF ]]; EN
  p Format.std_formatter test;
end
(** output
      test: [ "add" LEFTA
  [ SELF; "plus1plus"; SELF
  | SELF; "+"; SELF
  | SELF; "-"; SELF ]
| "mult" RIGHTA
  [ SELF; "plus1plus"; SELF
  | SELF; "*"; SELF
  | SELF; "/"; SELF ]
| "simple" NONA
  [ "("; SELF; ")"
  | INT ((_)) ]
| LEFTA
  [ SELF; "plus1plus"; SELF ] ]
*)
```

When you want to insert a new level before "mult"

```
let _ = begin
  EXTEND MGram  test: BEFORE "mult" [[x = SELF ; "plus1plus" ; y = SELF ]];
  END ;
  p Format.std_formatter test;
end
(** output
        test: [ "add" LEFTA
```

```
    [ SELF; "plus1plus"; SELF
    | SELF; "+"; SELF
    | SELF; "-"; SELF ]
  | LEFTA
    [ SELF; "plus1plus"; SELF ]
  | "mult" RIGHTA
    [ SELF; "plus1plus"; SELF
    | SELF; "*"; SELF
    | SELF; "/"; SELF ]
  | "simple" NONA
    [ "("; SELF; ")"
    | INT ((_)) ]
  | LEFTA
    [ SELF; "plus1plus"; SELF ] ]
  *)
```

9. Grammar example You can do some search in the toplevel as follows

```
se (FILTER _* "val" _* "expr" space+ ":" ) "Syntax" ;;
```

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

Code listed below is the expr parse tree

```
expr:
[ ";" LEFTA
  [ seq_expr ]

| "top" RIGHTA
  [ "RE_PCRE"; regexp
  | "REPLACE"; regexp; "->"; sequence
  | "SEARCH"; regexp; "->"; sequence
```

```
  | "MAP"; regexp; "->"; sequence
  | "COLLECT"; regexp; "->"; sequence
  | "COLLECTOBJ"; regexp
  | "SPLIT"; regexp
  | "REPLACE_FIRST"; regexp; "->"; sequence
  | "SEARCH_FIRST"; regexp; "->"; sequence
  | "MATCH"; regexp; "->"; sequence
  | "FILTER"; regexp
  | "CAPTURE"; regexp
  | "function"; OPT "|"; LIST1 regexp_match_case SEP "|"
  (* syntax extension by mikmatch*)
  | "parser"; OPT parser_ipatt; parser_case_list
  | "parser"; OPT parser_ipatt; parser_case_list
  | "let"; "try"; OPT "rec"; LIST1 let_binding SEP "and"; "in"; sequence;
    "with"; LIST1 lettry_case SEP "|"
 (*  syntax extension mikmatch
     let try a = raise Not_found in a with Not_found -> 24;; *)
  | "let"; LIDENT "view"; UIDENT _; "="; SELF; "in"; sequence
(*   view patterns *)
  | "let"; "module"; a_UIDENT; module_binding0; "in"; expr LEVEL ";"
  | "let"; "open"; module_longident; "in"; expr LEVEL ";"
  | "let"; OPT "rec"; binding; "in"; sequence
  | "if"; SELF; "then"; expr LEVEL "top"; "else"; expr LEVEL "top"
  | "if"; SELF; "then"; expr LEVEL "top"
  | "fun"; fun_def
  | "match"; sequence; "with"; "parser"; OPT parser_ipatt; parser_case_list
  | "match"; sequence; "with"; "parser"; OPT parser_ipatt; parser_case_list
  | "match"; sequence; "with"; OPT "|"; LIST1 regexp_match_case SEP "|"
  | "try"; SELF; "with"; OPT "|"; LIST1 regexp_match_case SEP "|"
  | "try"; sequence; "with"; match_case
  | "for"; a_LIDENT; "="; sequence; direction_flag; sequence; "do";
```

```
      do_sequence
    | "while"; sequence; "do"; do_sequence
    | "object"; opt_class_self_patt; class_structure; "end" ]
  | LEFTA
    [ "EXTEND"; extend_body; "END"
    | "DELETE_RULE"; delete_rule_body; "END"
    | "GDELETE_RULE"
    | "GEXTEND" ]


  (* operators *)
  | "," LEFTA
    [ SELF; ","; comma_expr ]

  | ":=" NONA
    [ SELF; ":="; expr LEVEL "top"
    | SELF; "<-"; expr LEVEL "top" ]

  | "||" RIGHTA
     [ SELF; infixop6; SELF ]

  | "&&" RIGHTA
    [ SELF; infixop5; SELF ]

  | "<" LEFTA
    [ SELF; infix operator (level 0) (comparison operators, and some others);
      SELF ]
  | "^" RIGHTA
    [ SELF; infix operator (level 1) (start with '^', '@'); SELF ]
  | "::" RIGHTA
    [ SELF; "::"; SELF ]
```

```
| "+" LEFTA
  [ SELF; infix operator (level 2) (start with '+', '-'); SELF ]
| "*" LEFTA
  [ SELF; "land"; SELF
  | SELF; "lor"; SELF
  | SELF; "lxor"; SELF
  | SELF; "mod"; SELF
  | SELF; infix operator (level 3) (start with '*', '/', '%'); SELF ]
| "**" RIGHTA
  [ SELF; "asr"; SELF
  | SELF; "lsl"; SELF
  | SELF; "lsr"; SELF
  | SELF; infix operator (level 4) (start with "**") (right assoc); SELF ]
| "unary minus" NONA
  [ "-"; SELF
  | "-."; SELF ]

(* apply *)
| "apply" LEFTA
  [ SELF; SELF
  | "assert"; SELF
  | "lazy"; SELF ]

| "label" NONA
  [ "~"; a_LIDENT
  | LABEL _; SELF
  | OPTLABEL _; SELF
  | "?"; a_LIDENT ]
| "." LEFTA
  [ SELF; "."; "("; SELF; ")"
  | SELF; "."; "["; SELF; "]"
```

```
      | SELF; "."; "{"; comma_expr; "}"
      | SELF; "."; SELF
      | SELF; "#"; label ]
    | "~-" NONA
      [ "!"; SELF
      | prefix operator (start with '!', '?', '~'); SELF ]
    | "simple" LEFTA
      [ "false"
      | "true"
      | "{"; TRY [ label_expr_list; "}" ]
      | "{"; TRY [ expr LEVEL "."; "with" ]; label_expr_list; "}"
      | "new"; class_longident
      | QUOTATION _
      | ANTIQUOT (("exp" | "" | "anti"), _)
      | ANTIQUOT ("`bool", _)
      | ANTIQUOT ("tup", _)
      | ANTIQUOT ("seq", _)
      | "`"; a_ident
      | "["; "]"
      | "["; sem_expr_for_list; "]"
      | "[|"; "|]"
      | "[|"; sem_expr; "|]"
      | "{<"; ">}"
      | "{<"; field_expr_list; ">}"
      | "begin"; "end"
      | "begin"; sequence; "end"
      | "("; ")"
      | "("; "module"; module_expr; ")"
      | "("; "module"; module_expr; ":"; package_type; ")"
      | "("; SELF; ";"; ")"
      | "("; SELF; ";"; sequence; ")"
```

```
| "("; SELF; ":"; ctyp; ")"
| "("; SELF; ":"; ctyp; ":>"; ctyp; ")"
| "("; SELF; ":>"; ctyp; ")"
| "("; SELF; ")"
| stream_begin; stream_end
| stream_begin; stream_expr_comp_list; stream_end
| stream_begin; stream_end
| stream_begin; stream_expr_comp_list; stream_end
| a_INT
| a_INT32
| a_INT64
| a_NATIVEINT
| a_FLOAT
| a_STRING
| a_CHAR
| TRY module_longident_dot_lparen; sequence; ")"
| TRY val_longident ] ]
```

A syntax extension of `let try`

```
let try a = 3 in true with Not_found -> false || false;;
true
```

First, it uses start parser to parse *let try a = 3 in true with Not_found -> false*, then it calls the cont parser, and the next level cont parser, etc, and then it succeeds. This also applies to "apply" level.

A tiny extension(you modify the Camlp4.PreCast.Gram, it will be reflected on the fly)

```
open Camlp4.PreCast
let env = ref []
(** Toploop.toplevel_env *)
(** sucks, in the toplevel, it's really hard to roll back cause, all
```

```
    your programs following are affected , horrible *)
let _ = begin
  let _loc = Loc.ghost in
  EXTEND Gram Syntax.expr:
    LEVEL "simple" [[x = LIDENT -> List.assoc x !env ]] ; END ;
  env := ["x", <:expr< 3 >> ]
end

let y = 4 in let a = x + y in a;;

(** Error: Camlp4: Uncaught exception: Not_found
   first y,a  is pat
   second y results in an exception
*)
(** DELETE_RULE Gram Syntax.expr: LIDENT    END ;; *)


(** NOT supported yet
let add_infix lev op =
    EXTEND Gram
      Syntax.expr :
      LEVEL $lev$
      [[ x = SELF ;  $op$ ; y = SELF ->
      <:expr< $lid:op$ $x$ $y$ >>]]; END  ;;
*)
```

10. when two rules overlapping, the EXTEND statement replaces the old version by the new one and displays a warning.

```
se (FILTER _* "warning") "Syntax"
```

```
type warning = Loc.t -> string -> unit
val default_warning : warning
```

```
    val current_warning : warning ref
    val print_warning : warning
```

4.7 Rewrite of Jake's blog

Jake's blog is a very comprehensive tutorial for camlp4 introduction.

4.7.1 Part1

Easy to experiment in the toplevel, using my previous **oco**,

```
open Camlp4.PreCast ;;
let _loc = Loc.ghost ;;
(**
   blabla...
   An idea, how about writing another pretty printer,
   the printer is awful*)
```

4.7.2 Part2

Just ast transform, easy to experiment in toplevel as well.

```
(* #require "camlp4.gramlib";; *)
open Camlp4.PreCast
open BatPervasives
let cons = ["A"; "B";"C"] and _loc = Loc.ghost ;;
let tys = Ast.tyOr_of_list
  (List.map (fun str -> <:ctyp< $uid:str$ >>) cons);;

(*
val tys : Camlp4.PreCast.Ast.ctyp =
  Camlp4.PreCast.Ast.TyOr (<abstr>,
   Camlp4.PreCast.Ast.TyId (<abstr>, Camlp4.PreCast.Ast.IdUid (<abstr>, "A")),
```

```
   Camlp4.PreCast.Ast.TyOr (<abstr>,
    Camlp4.PreCast.Ast.TyId (<abstr>,
     Camlp4.PreCast.Ast.IdUid (<abstr>, "B")),
    Camlp4.PreCast.Ast.TyId (<abstr>,
     Camlp4.PreCast.Ast.IdUid (<abstr>, "C"))))
*)
(** here you can better understand what ctyp really means, a type
expression, not a top-level struct, cool
*)
let verify = <:ctyp< A |B |C>>;;

let _ = begin
  print_bool (verify = tys);
end

(*
  true
*)

let type_def = <:str_item< type t = $tys$ >>
(*
val type_def : Camlp4.PreCast.Ast.str_item =
  Camlp4.PreCast.Ast.StTyp (<abstr>,
   Camlp4.PreCast.Ast.TyDcl (<abstr>, "t", [],
    Camlp4.PreCast.Ast.TyOr (<abstr>,
     Camlp4.PreCast.Ast.TyId (<abstr>,
      Camlp4.PreCast.Ast.IdUid (<abstr>, "A")),
     Camlp4.PreCast.Ast.TyOr (<abstr>,
      Camlp4.PreCast.Ast.TyId (<abstr>,
       Camlp4.PreCast.Ast.IdUid (<abstr>, "B")),
      Camlp4.PreCast.Ast.TyId (<abstr>,
```

```
       Camlp4.PreCast.Ast.IdUid (<abstr>, "C")))),
    []))
*)

let _ =  begin
  Printers.OCaml.print_implem type_def ;
end
(*
  type t = A | B | C
*)


(** always ambiguous when manipulating ast using original syntax
    recommend using revised syntx
*)
let verify2 = <:str_item< type t =  [ A | B | C ] >>;;
(*
val verify2 : Camlp4.PreCast.Ast.str_item =
  Camlp4.PreCast.Ast.StTyp (<abstr>,
   Camlp4.PreCast.Ast.TyDcl (<abstr>, "t", [],
    Camlp4.PreCast.Ast.TySum (<abstr>,
     Camlp4.PreCast.Ast.TyOr (<abstr>,
      Camlp4.PreCast.Ast.TyOr (<abstr>,
       Camlp4.PreCast.Ast.TyId (<abstr>,
        Camlp4.PreCast.Ast.IdUid (<abstr>, "A")),
       Camlp4.PreCast.Ast.TyId (<abstr>,
        Camlp4.PreCast.Ast.IdUid (<abstr>, "B"))),
      Camlp4.PreCast.Ast.TyId (<abstr>,
       Camlp4.PreCast.Ast.IdUid (<abstr>, "C")))),
    []))
*)
```

```
let _ = begin
  print_bool (verify2 = type_def);
  Printers.OCaml.print_implem verify2
end

(*
  false
  type t = | A | B | C;;
*)

let match_case =
  List.map
    (fun c -> <:match_case< $uid:c$ -> $`str:c$ >>) cons
    |> Ast.mcOr_of_list ;;
(*
  Camlp4.PreCast.Ast.McOr (<abstr>,
   Camlp4.PreCast.Ast.McArr (<abstr>,
    Camlp4.PreCast.Ast.PaId (<abstr>,
     Camlp4.PreCast.Ast.IdUid (<abstr>, "A")),
    Camlp4.PreCast.Ast.ExNil <abstr>,
    Camlp4.PreCast.Ast.ExStr (<abstr>, "A")),
   Camlp4.PreCast.Ast.McOr (<abstr>,
    Camlp4.PreCast.Ast.McArr (<abstr>,
     Camlp4.PreCast.Ast.PaId (<abstr>,
      Camlp4.PreCast.Ast.IdUid (<abstr>, "B")),
     Camlp4.PreCast.Ast.ExNil <abstr>,
     Camlp4.PreCast.Ast.ExStr (<abstr>, "B")),
    Camlp4.PreCast.Ast.McArr (<abstr>,
     Camlp4.PreCast.Ast.PaId (<abstr>,
      Camlp4.PreCast.Ast.IdUid (<abstr>, "C")),
     Camlp4.PreCast.Ast.ExNil <abstr>,
```

```
     Camlp4.PreCast.Ast.ExStr (<abstr>, "C"))))
*)
let to_string = <:expr< fun [ $match_case$ ] >>;;
(*
    Camlp4.PreCast.Ast.ExFun (<abstr>,
   Camlp4.PreCast.Ast.McOr (<abstr>,
    Camlp4.PreCast.Ast.McArr (<abstr>,
     Camlp4.PreCast.Ast.PaId (<abstr>,
      Camlp4.PreCast.Ast.IdUid (<abstr>, "A")),
     Camlp4.PreCast.Ast.ExNil <abstr>,
     Camlp4.PreCast.Ast.ExStr (<abstr>, "A")),
    Camlp4.PreCast.Ast.McOr (<abstr>,
     Camlp4.PreCast.Ast.McArr (<abstr>,
      Camlp4.PreCast.Ast.PaId (<abstr>,
       Camlp4.PreCast.Ast.IdUid (<abstr>, "B")),
      Camlp4.PreCast.Ast.ExNil <abstr>,
      Camlp4.PreCast.Ast.ExStr (<abstr>, "B")),
     Camlp4.PreCast.Ast.McArr (<abstr>,
      Camlp4.PreCast.Ast.PaId (<abstr>,
       Camlp4.PreCast.Ast.IdUid (<abstr>, "C")),
      Camlp4.PreCast.Ast.ExNil <abstr>,
      Camlp4.PreCast.Ast.ExStr (<abstr>, "C")))))
*)

let pim = Printers.OCaml.print_implem

let _ = begin
  pim <:str_item< let a = $to_string$ in a >>;
end
(*
  let a = function | A -> "A" | B -> "B" | C -> "C" in a;;
```

```
*)

let match_case2 = List.map
  (fun c -> <:match_case< $`str:c$ -> $uid:c$
    >>) cons|> Ast.mcOr_of_list


let _ = begin
  pim  <:str_item<let f = fun [ $match_case2$ ] in f  >>;
  pim <:str_item<let f = fun [ $match_case2$ | _ -> invalid_arg "haha" ] in f >>;
end

(*

let f = function | "A" -> A | "B" -> B | "C" -> C in f;;
let f = function | "A" -> A | "B" -> B | "C" -> C | _ -> invalid_arg "haha"
in f;;

*)
```

Anyother way to verify? The output of printers does not seem to guarantee its correctness. When you do antiquotation, in the cases of inserting an AST rather than a string, usually you *do not* need tags, when you inserting a string, probably you *need it*.

4.7.3 Part3 : Quotations in Depth

```
[`QUOTATION x -> Quotation.expand _loc x Quotation.DynAst.expr_tag ]
```

The `QUOTATION token contains a record including the body of the quotation and the `tag`. The record is passed off to the Quotation module to be expanded. The expander parses the quotation string starting at some non-terminal(you specified), then runs the result through the antiquotation expander

```
|`ANTIQUOT (``exp'' | ``'' | ``anti'' as n) s ->
<:expr< $anti:make_anti ~c:"expr" n s $>>
```

The antiquotation creates `a special AST node` to hold the body of the antiquotation, each type in the AST has a constructor (`ExAnt, TyAnt`, etc.) `c` means context here.

Here we grep `Ant`, and the output is as follows

```
27 matches for "Ant" in buffer: Camlp4Ast.partial.ml
     5:     | BAnt of string ]
     9:     | ReAnt of string ]
    13:     | DiAnt of string ]
    17:     | MuAnt of string ]
    21:     | PrAnt of string ]
    25:     | ViAnt of string ]
    29:     | OvAnt of string ]
    33:     | RvAnt of string ]
    37:     | OAnt of string ]
    41:     | LAnt of string ]
    47:     | IdAnt of loc and string (* $s$ *) ]
    87:     | TyAnt of loc and string (* $s$ *)
    93:     | PaAnt of loc and string (* $s$ *)
   124:     | ExAnt of loc and string (* $s$ *)
   202:     | MtAnt of loc and string (* $s$ *) ]
   231:     | SgAnt of loc and string (* $s$ *) ]
   244:     | WcAnt of loc and string (* $s$ *) ]
   251:     | BiAnt of loc and string (* $s$ *) ]
   258:     | RbAnt of loc and string (* $s$ *) ]
   267:     | MbAnt of loc and string (* $s$ *) ]
   274:     | McAnt of loc and string (* $s$ *) ]
   290:     | MeAnt of loc and string (* $s$ *) ]
   321:     | StAnt of loc and string (* $s$ *) ]
   337:     | CtAnt of loc and string ]
   352:     | CgAnt of loc and string (* $s$ *) ]
   372:     | CeAnt of loc and string ]
   391:     | CrAnt of loc and string (* $s$ *) ];
```

ANTIQUOTATION example

```
<:expr< $int: "4"$ >>;;
- : Camlp4.PreCast.Ast.expr = Camlp4.PreCast.Ast.ExInt (<abstr>, "4")
<:expr< $`int: 4$ >>;; (** the same result *)
- : Camlp4.PreCast.Ast.expr = Camlp4.PreCast.Ast.ExInt (<abstr>, "4")
<:expr< $`flo:4.1323243232$ >>;;
- : Camlp4.PreCast.Ast.expr = Camlp4.PreCast.Ast.ExFlo (<abstr>, "4.1323243232")
# <:expr< $flo:"4.1323243232"$ >>;;
- : Camlp4.PreCast.Ast.expr = Camlp4.PreCast.Ast.ExFlo (<abstr>, "4.1323243232")
(** maybe the same for flo *)
```

```
  match_case:
    [ [ "["; l = LIST0 match_case0 SEP "|"; "]" -> Ast.
        mcOr_of_list l
      | p = ipatt; "->"; e = expr -> <:match_case< $p$ -> $e$ >> ]
          ]
  ;
  match_case0:
    [ [ `ANTIQUOT ("match_case"|"list" as n) s ->
          <:match_case< $anti:mk_anti ~c:"match_case" n s$ >>
      | `ANTIQUOT (""|"anti" as n) s ->
          <:match_case< $anti:mk_anti ~c:"match_case" n s$ >>
      | `ANTIQUOT (""|"anti" as n) s; "->"; e = expr ->
          <:match_case< $anti:mk_anti ~c:"patt" n s$ -> $e$ >>
      | `ANTIQUOT (""|"anti" as n) s; "when"; w = expr; "->"; e =
         expr ->
          <:match_case< $anti:mk_anti ~c:"patt" n s$ when $w$ ->
             $e$ >>
      | p = patt_as_patt_opt; w = opt_when_expr; "->"; e = expr ->
          <:match_case< $p$ when $w$ -> $e$ >>
    ] ]
```

You can see that `match_case0`, if we use the list antiquotation, the first case in `match_case0` returns an antiquotation with tag `listmatch_case`,and we get the following expansion

```
value antiquot_expander = object
  inherit Ast.map as super;
  method patt = fun
      [ <:patt@_loc< $anti:s$ >> | <:patt@_loc< $str:s$ >> as p ->
          let mloc _loc = MetaLoc.meta_loc_patt _loc _loc in
          handle_antiquot_in_string s p TheAntiquotSyntax.parse_patt _loc (fun n p
```

```
          match n with
          [ "antisig_item" -> <:patt< Ast.SgAnt $mloc _loc$ $p$ >>
          | "antistr_item" -> <:patt< Ast.StAnt $mloc _loc$ $p$ >>
          | "antictyp" -> <:patt< Ast.TyAnt $mloc _loc$ $p$ >>
          | "antipatt" -> <:patt< Ast.PaAnt $mloc _loc$ $p$ >>
          | "antiexpr" -> <:patt< Ast.ExAnt $mloc _loc$ $p$ >>
          | "antimodule_type" -> <:patt< Ast.MtAnt $mloc _loc$ $p$ >>
          | "antimodule_expr" -> <:patt< Ast.MeAnt $mloc _loc$ $p$ >>
          | "anticlass_type" -> <:patt< Ast.CtAnt $mloc _loc$ $p$ >>
          | "anticlass_expr" -> <:patt< Ast.CeAnt $mloc _loc$ $p$ >>
          | "anticlass_sig_item" -> <:patt< Ast.CgAnt $mloc _loc$ $p$ >>
          | "anticlass_str_item" -> <:patt< Ast.CrAnt $mloc _loc$ $p$ >>
          | "antiwith_constr" -> <:patt< Ast.WcAnt $mloc _loc$ $p$ >>
          | "antibinding" -> <:patt< Ast.BiAnt $mloc _loc$ $p$ >>
          | "antirec_binding" -> <:patt< Ast.RbAnt $mloc _loc$ $p$ >>
          | "antimatch_case" -> <:patt< Ast.McAnt $mloc _loc$ $p$ >>
          | "antimodule_binding" -> <:patt< Ast.MbAnt $mloc _loc$ $p$ >>
          | "antiident" -> <:patt< Ast.IdAnt $mloc _loc$ $p$ >>
          | _ -> p ])
          | p -> super#patt p ];
    method expr = fun
      [ <:expr@_loc< $anti:s$ >> | <:expr@_loc< $str:s$ >> as e ->
          let mloc _loc = MetaLoc.meta_loc_expr _loc _loc in
          handle_antiquot_in_string s e TheAntiquotSyntax.parse_expr _loc (fun n e
          match n with
          [ "`int" -> <:expr< string_of_int $e$ >>
          | "`int32" -> <:expr< Int32.to_string $e$ >>
          | "`int64" -> <:expr< Int64.to_string $e$ >>
          | "`nativeint" -> <:expr< Nativeint.to_string $e$ >>
          | "`flo" -> <:expr< Camlp4_import.Oprint.float_repres $e$ >>
          | "`str" -> <:expr< Ast.safe_string_escaped $e$ >>
```

```
| "`chr" -> <:expr< Char.escaped $e$ >>
| "`bool" -> <:expr< Ast.IdUid $mloc _loc$ (if $e$ then "True" else "F
| "liststr_item" -> <:expr< Ast.stSem_of_list $e$ >>
| "listsig_item" -> <:expr< Ast.sgSem_of_list $e$ >>
| "listclass_sig_item" -> <:expr< Ast.cgSem_of_list $e$ >>
| "listclass_str_item" -> <:expr< Ast.crSem_of_list $e$ >>
| "listmodule_expr" -> <:expr< Ast.meApp_of_list $e$ >>
| "listmodule_type" -> <:expr< Ast.mtApp_of_list $e$ >>
| "listmodule_binding" -> <:expr< Ast.mbAnd_of_list $e$ >>
| "listbinding" -> <:expr< Ast.biAnd_of_list $e$ >>
| "listbinding;" -> <:expr< Ast.biSem_of_list $e$ >>
| "listrec_binding" -> <:expr< Ast.rbSem_of_list $e$ >>
| "listclass_type" -> <:expr< Ast.ctAnd_of_list $e$ >>
| "listclass_expr" -> <:expr< Ast.ceAnd_of_list $e$ >>
| "listident" -> <:expr< Ast.idAcc_of_list $e$ >>
| "listctypand" -> <:expr< Ast.tyAnd_of_list $e$ >>
| "listctyp;" -> <:expr< Ast.tySem_of_list $e$ >>
| "listctyp*" -> <:expr< Ast.tySta_of_list $e$ >>
| "listctyp|" -> <:expr< Ast.tyOr_of_list $e$ >>
| "listctyp," -> <:expr< Ast.tyCom_of_list $e$ >>
| "listctyp&" -> <:expr< Ast.tyAmp_of_list $e$ >>
| "listwith_constr" -> <:expr< Ast.wcAnd_of_list $e$ >>
(* interesting bits *)
| "listmatch_case" -> <:expr< Ast.mcOr_of_list $e$ >>
| "listpatt," -> <:expr< Ast.paCom_of_list $e$ >>
| "listpatt;" -> <:expr< Ast.paSem_of_list $e$ >>
| "listexpr," -> <:expr< Ast.exCom_of_list $e$ >>
| "listexpr;" -> <:expr< Ast.exSem_of_list $e$ >>
| "antisig_item" -> <:expr< Ast.SgAnt $mloc _loc$ $e$ >>
| "antistr_item" -> <:expr< Ast.StAnt $mloc _loc$ $e$ >>
| "antictyp" -> <:expr< Ast.TyAnt $mloc _loc$ $e$ >>
```

```
              | "antipatt" -> <:expr< Ast.PaAnt $mloc _loc$ $e$ >>
              | "antiexpr" -> <:expr< Ast.ExAnt $mloc _loc$ $e$ >>
              | "antimodule_type" -> <:expr< Ast.MtAnt $mloc _loc$ $e$ >>
              | "antimodule_expr" -> <:expr< Ast.MeAnt $mloc _loc$ $e$ >>
              | "anticlass_type" -> <:expr< Ast.CtAnt $mloc _loc$ $e$ >>
              | "anticlass_expr" -> <:expr< Ast.CeAnt $mloc _loc$ $e$ >>
              | "anticlass_sig_item" -> <:expr< Ast.CgAnt $mloc _loc$ $e$ >>
              | "anticlass_str_item" -> <:expr< Ast.CrAnt $mloc _loc$ $e$ >>
              | "antiwith_constr" -> <:expr< Ast.WcAnt $mloc _loc$ $e$ >>
              | "antibinding" -> <:expr< Ast.BiAnt $mloc _loc$ $e$ >>
              | "antirec_binding" -> <:expr< Ast.RbAnt $mloc _loc$ $e$ >>
              | "antimatch_case" -> <:expr< Ast.McAnt $mloc _loc$ $e$ >>
              | "antimodule_binding" -> <:expr< Ast.MbAnt $mloc _loc$ $e$ >>
              | "antiident" -> <:expr< Ast.IdAnt $mloc _loc$ $e$ >>
              | _ -> e ])
       | e -> super#expr e ];
```

Here we see the ambiguity of original syntax,

```
<< type t = [ $list:List.map (fun c -> <:ctyp< $uid:c$ >>)$]  >>
```

In original syntax, it does not know it's variant context, or just type synonm. (you can add a constructor to make it clear)

4.7.4 Part4 parsing ocaml itself using camlp4

We have to use revised syntax here, because when using quasiquotation, it has ambiguity to get the needed part, revised syntax was designed to reduce the ambiguity.

The following code is a greate file parsing ocaml itself. Do not use MakeSyntax below, since it will introduce unnecessary abstraction type, which makes sharing code very difficult

```
open Camlp4.PreCast ;
open BatPervasives ;
```

```
(** My own syntax *)
module MySyntax = Camlp4.OCamlInitSyntax.Make Ast Gram Quotation ;

(** load r parser *)
module M = Camlp4OCamlRevisedParser.Make MySyntax ;

(** load quotation parser *)
module M4 = Camlp4QuotationExpander.Make MySyntax ;


(** in toplevel, I did not find a way to introduce such module
    because it will change the state
*)

(**
module N = Camlp4OCamlParser.Make MySyntax ;
load o parser
*)

value my_parser = MySyntax.parse_implem;
value str_items_of_file file_name =
  file_name
  |> open_in
  |> BatStream.of_input
  |> my_parser (Loc.mk file_name)
  |> flip Ast.list_of_str_item [] ;

(** it has ambiguity in original syntax, so pattern match
    will be more natural in revised syntax
*)
value rec do_str_item str_item tags =
```

```
  match str_item with
      [ <:str_item< value $rec:_$ $binding$ >> ->
        let bindings = Ast.list_of_binding binding []
        in List.fold_right do_binding bindings tags
      |_ -> tags ]
and do_binding bi tags = match bi with
  [ <:binding@loc< $lid:lid$ = $_$ >> ->
    let line = Loc.start_line loc in
    let off = Loc.start_off loc in
    let pre = "let " ^ lid in
    [(pre,lid,line,off) :: tags ]
  | _ -> tags ];


value do_fn file_name =
    file_name
    |> str_items_of_file
    |> List.map (flip do_str_item [])
    |> List.concat ;
(**use MSyntax.parse_implem*)
value _ =
  do_fn "otags.ml"
  |> List.iter (fun (a, b, c, d) -> Printf.printf "%s-%s %d-%d \n" a b c d)  ;


(* output
  let my_parser-my_parser 22-469
  let str_items_of_file-str_items_of_file 23-510
  let do_str_item-do_str_item 33-779
  let do_binding-do_binding 39-1012
  let do_fn-do_fn 48-1256
```

```
*)

let sig =
  let str = eval "module X = Camlp4.PreCast ;;"
  and _loc = Loc.ghost in
  Stream.of_string str |> Syntax.parse_interf  _loc ;;

open Camlp4.PreCast.Syntax.Ast

(* output
SgMod (<abstr>, "X",
 MtSig (<abstr>,
  SgSem (<abstr>,
   SgSem (<abstr>,
    SgTyp (<abstr>,
     TyDcl (<abstr>, "camlp4_token", [],
      TyMan (<abstr>,
       TyId (<abstr>,
        IdAcc (<abstr>,
         IdAcc (<abstr>, IdUid (<abstr>, "Camlp4"), IdUid (<abstr>, "Sig")),
         IdLid (<abstr>, "camlp4_token"))),
       TySum (<abstr>,
        TyOr (<abstr>,
         TyOr (<abstr>,
          TyOr (<abstr>,
           TyOr (<abstr>,
            TyOr (<abstr>,
             TyOr (<abstr>,
              TyOr (<abstr>,
               TyOr (<abstr>,
                TyOr (<abstr>,
                 TyOr (<abstr>,
```

```
TyOr (<abstr>,
 TyOr (<abstr>,
  TyOr (<abstr>,
   TyOr (<abstr>,
    TyOr (<abstr>,
     TyOr (<abstr>,
      TyOr (<abstr>,
       TyOr (<abstr>,
        TyOr (<abstr>,
         TyOr (<abstr>,
          TyOf (<abstr>,
           TyId (<abstr>, IdUid (<abstr>, "KEYWORD")),
           TyId (<abstr>, IdLid (<abstr>, "string"))),
          TyOf (<abstr>,
           TyId (<abstr>, IdUid (<abstr>, "SYMBOL")),
           TyId (<abstr>, IdLid (<abstr>, "string")))),
         TyOf (<abstr>,
          TyId (<abstr>, IdUid (<abstr>, "LIDENT")),
          TyId (<abstr>, IdLid (<abstr>, "string")))),
        TyOf (<abstr>,
         TyId (<abstr>, IdUid (<abstr>, "UIDENT")),
         TyId (<abstr>, IdLid (<abstr>, "string")))),
       TyOf (<abstr>,
        TyId (<abstr>, IdUid (<abstr>, "ESCAPED_IDENT")),
        TyId (<abstr>, IdLid (<abstr>, "string")))),
      TyOf (<abstr>,
       TyId (<abstr>, IdUid (<abstr>, "INT")),
       TyAnd (<abstr>,
        TyId (<abstr>, IdLid (<abstr>, "int")),
        TyId (<abstr>, IdLid (<abstr>, "string"))))),
     TyOf (<abstr>,
```

```
                        TyId (<abstr>, IdUid (<abstr>, "INT32")), ...)),
                      ...),
                     ...),
                    ...),
                   ...),
                  ...),
                 ...),
                ...),
               ...),
              ...),
             ...),
            ...),
           ...),
          ...),
         ...))),
      ...)),
    ...),
   ...)))
*)
```

4.7.5 Part5 Structure Item Filters

Because I use revised syntax, and take a reference of the documenation, my ast filter is much nicer than jaked's. the documentation of quasiquotation from the wiki page is quite helpful

```
(* open BatPervasives; *)

open Camlp4.PreCast;
open Printf;
(* open Batteries; *)
(* value pim = Printers.OCaml.print_implem ; *)
```

```
open Util;

module Make (AstFilters : Camlp4.Sig.AstFilters) = struct
  open AstFilters;
  value code_of_con_names name cons _loc =
    let match_cases = cons
      |> List.map (fun str -> <:match_case< $uid:str$ -> $str:str$ >>)
      (* |> Ast.mcOr_of_list *)
    in
    let reverse_cases = cons
      |> List.map (fun con -> <:match_case< $str:con$ -> $uid:con$ >>)
      (* |> Ast.mcOr_of_list *)
    in
    <:str_item<
      value $lid:(name^"_to_string") $ =
          fun [ $list:match_cases$ ]
      ;
      value $lid:(name^"_of_string") $ =
          fun [ $list:reverse_cases$ | x -> invalid_arg x ]
      ; >>;


 (** idea, view patterns are fit here, try it later *)
 value rec filter str_item = match str_item with
     [
       <:str_item@_loc< type $lid:tid$ = [ $t$ ]>> -> begin
         try
           let ctys = Ast.list_of_ctyp t [] in
           let con_names =
             List.map (fun [ <:ctyp< $uid:c$>> -> c
                           | x -> raise Not_found ]) ctys in
```

```
          let code = code_of_con_names tid con_names _loc in
          begin
            prerr_string "generating code right now";
            <:str_item< $str_item$; $code$>>
          end
        with
            [exn -> begin
              prerr_string (sprintf "%s\n : error \n" tid);
              raise Not_found;
            end ]
      end
    | x -> begin
      x
    end
    ] ;
  AstFilters.register_str_item_filter filter;
end ;

module Id = struct
  value name = "pa_filter";
  value version = "0.1";
end ;

value _ =
  let module M = Camlp4.Register.AstFilter Id Make in
  ();
```

For locally used syntax extension, I found write some tiny bits ocamlbuild code pretty convenient. In `myocamlbuild.ml`, only needs to append some code like this

```
let apply  plugin = begin
```

```
Default.before_options +> before_options;
Default.after_rules +> after_rules;
plugin ();
dispatch begin function
  | Before_options -> begin
    List.iter (fun f -> f () ) !before_options;
  end
  | After_rules -> begin
    List.iter (fun f -> f ()) !after_rules;
  end
```

The tags file then will be like this

```
<pa_filter.{ml}> : pp(camlp4rf)
<pa_filter.{cmo}> : use_camlp4
<test_filter.ml> : camlp4o, use_filter
```

Using Register Filter has some limitations, like it first parse the whole file, and then transform each `structure item` one by one, so the previously generated code will `not have` an effect on the later code. This is probably what not you want.

You syntax extension may depends on other modules, make sure your `pa_xx.cma` contains all the modules statically. You can write a `pa_xx.mllib`, or link the module to `cma` file by hand.

For instance, you `pa_filter.cma` depends on `Util`, then you will

`ocalmc -a pa_filter.cmo util.cmo -o pa_filer.cma`

then you could use `camlp4o -parser pa_filter.cma`, it works.

If you write `pa_xx.mllib` file, it would be something like

```
pa_filter
util
```

If you want to use other libraries to write syntax extension, make sure you link **all** libraries, including recursive dependency, i.e, the require field of batteries.

```
ocamlc -a  -I +num -I `ocamlfind query batteries` nums.cma unix.cma
bigarray.cma str.cma batteries.cma pa_filter.cma -o x.cma
```

You must link all the libraries, even you don't need it at all. This is the defect of the OCaml compiler. `-linkall` here links submodules, recursive linking needs you say it clearly.

We can also test our filter seriously as follows `camlp4of -parser _build/filter.cmo filter_t`

By the **lift filter** you can see its **internal representation**, textual code does not gurantee its correctness, but the AST representation could gurantee its correctness.

Built in filters as follows :

(a) fold map

```
class x = Camlp4MapGenerator.generated ;
class x = Camlp4FoldGenerator.generated ;
```

(b) meta

lifting function from a type definition – these functions are what *Camlp4AstLifter uses* to lift the AST, and also how *quotations are implemented*

(c) LocationStripper (replace location with Loc.ghost)

might be useful when you compare two asts? YES! idea? how to use lifter at toplevel, how to beautify our code, without the horribling output? (I mean, the qualified name is horrible)

(d) Camlp4Profiler

inserts profiling code

(e) Camlp4TrashRemover

(f) Camlp4ExceptionTracer

4.7.6 Part6 Extensible Parser (moved to extensible parser part)

4.7.7 Part7 Revised Syntax

Revised syntax provides more context in the form of extra brackets etc. so that antiquotation works more smoothly.

4.7.8 Part8, 9 Quotation

(a) Quotation.add quotation_expander

```
se (FILTER _* "expand_fun") "Quotation";;

type 'a expand_fun = Ast.loc -> string option -> string -> 'a
val add : string -> 'a DynAst.tag -> 'a expand_fun -> unit
val find : string -> 'a DynAst.tag -> 'a expand_fun
```

other useful functions

```
type 'a expand_fun = Ast.loc -> string option -> string -> 'a
val add : string -> 'a DynAst.tag -> 'a expand_fun -> unit
val find : string -> 'a DynAst.tag -> 'a expand_fun
val default : string ref   (* default quotations *)
val parse_quotation_result :
      (Ast.loc -> string -> 'a) ->
      Ast.loc -> Camlp4.Sig.quotation -> string -> string -> 'a
val translate : (string -> string) ref
val expand : Ast.loc -> Camlp4.Sig.quotation -> 'a DynAst.tag -> 'a
val dump_file : string option ref
```

in previous camlp4, Quotation provides a string to string transformation, then it default uses Syntax.expr or Syntax.patt to parse the returned string. following drawbacks

- needs a **more** parsing phase
- the resulting string may be syntactically incorrect, difficult to **debug**

(b) quotation expander

when without antiquotaions, a parser is enought, other things are quite mechanical

```
open Camlp4.PreCast
module Jq_ast = struct
  type float' = float
  type t =
      Jq_null
    |Jq_bool of bool
    |Jq_number of float'
    |Jq_string of string
    |Jq_array of t list
    |Jq_object of (string*t) list
end
include Jq_ast
module MetaExpr = struct
  (** the generator scans all the types defined in the current module
      then generate code for the last-appearing recursive bundle
  *)
  let meta_float' _loc f = <:expr< $`flo:f$ >>
  include Camlp4Filters.MetaGeneratorExpr(Jq_ast)
  (* due to this can not run in toplevel *)
end
module MetaPatt = struct
  let meta_float' _loc f = <:patt< $`flo:f$ >>
  include Camlp4Filters.MetaGeneratorPatt(Jq_ast)
end
module MGram = MakeGram(Lexer)
```

```
let json_parser = MGram.Entry.mk "json"
  EXTEND MGram
  GLOBAL : json_parser ;
  json_parser :
    [["null" -> Jq_null
     |"true" -> Jq_bool true
     |"false" -> Jq_bool false
     | n = [x = INT -> x | y = FLOAT -> y  ] -> Jq_number (float_of_string n )
     | s = STRING -> Jq_string s
     | "["; xs = LIST0 SELF SEP "," ; "]" -> Jq_array xs
     | "{"; kvs = LIST0 [s = STRING; ":"; v = json_parser -> (s,v)] SEP ",";
       "}" -> Jq_object kvs
     ]] ; END
let json_eoi = MGram.Entry.mk "json_eoi"
  EXTEND MGram
  GLOBAL: json_eoi ;
  json_eoi : [[x = json_parser ; EOI -> x ]] ; END
let test =
  MGram.parse_string json_eoi (Loc.mk "<string>")
    "[true,false]"
```

Mechanical installation to get a quotation expander

```
module Q = Syntax.Quotation
(* #directory "/Users/bob/SourceCode/OCaml/Parsing/camlp4/_build";; *)
(* camlp4of -filter meta json.ml -printer o *)
let (|>) x f = f x
let parse_quot_string _loc s =
  MGram.parse_string  json_eoi _loc s
let expand_expr _loc _ s =
  s
  |> parse_quot_string _loc
```

```
  |> MetaExpr.meta_t _loc

(* to make it able to appear in the toplevel *)
let expand_str_item _loc _ s =
  (**insert an expression as str_item *)
   <:str_item@_loc< $exp: expand_expr _loc None s $ >>
let expand_patt _loc _ s  =
  s
  |> parse_quot_string _loc
  |> MetaPatt.meta_t _loc

let _  =
  Q.add "json" Q.DynAst.expr_tag expand_expr ;
  Q.add "json" Q.DynAst.patt_tag expand_patt ;
  Q.add "json" Q.DynAst.str_item_tag expand_str_item ;
  Q.default := "json"

(** make quotation from a parser *)
let install_quotation my_parser (me,mp) name =
  let module Q = Syntax.Quotation in
  let expand_expr _loc _ s = s |>  my_parser _loc |> me _loc in
  let expand_str_item _loc _ s =  <:str_item@_loc< $exp: expand_expr
  _loc None s $>> in
  let expand_patt _loc _ s = s |> my_parser _loc |> mp _loc in
  Q.add name Q.DynAst.expr_tag expand_expr ;
  Q.add "json" Q.DynAst.patt_tag expand_patt ;
  Q.add "json" Q.DynAst.str_item_tag expand_str_item
```

```
val install_quotation :
  (Camlp4.PreCast.Ast.loc -> string -> 'a) ->
  (Camlp4.PreCast.Ast.loc -> 'a -> Camlp4.PreCast.Ast.expr) *
```

```
(Camlp4.PreCast.Ast.loc -> 'a -> Camlp4.PreCast.Ast.patt) ->
    string -> unit =
<fun>
```

```
"json.ml" : pp(camlp4of -filter meta)
<json.{cmo,byte,native}> : pkg_dynlink, use_camlp4_full
```

so in the toplevel

```
#directory "/Users/bob/SourceCode/OCaml/Parsing/camlp4/_build";;
#load "json.cmo" ;
open Json;  (* for Jq_ast module, you can find other ways to work
around this *)
```

```
 << [ 3 ,4 ]>>;;
- : Json.Jq_ast.t = Json.Jq_ast.Jq_array [Json.Jq_ast.Jq_number 3.; Json.Jq_ast.Jq_number
4.]
```

(c) antiquotation expander

the meta filter treat any other constructor **ending in Ant** specially

instead of

```
|Jq_Ant(loc,s) -> <:expr< Jq_Ant ($meta_loc loc$, $meta_string s$) >>
```

they have

```
|Jq_Ant(loc,s) -> ExAnt(loc,s)
```

Instead of lifting the constructor, they translate it directly to ExAnt or PaAnt.

Attention, there is no semi or comma required in GLOBAL list, GLOBAL: json_eoi json ; (just whitespace)

```
open Camlp4.PreCast
module Jq_ast = struct
  type float' = float
  type t =
      Jq_null
    |Jq_bool of bool
    |Jq_number of float'
    |Jq_string of string

    |Jq_array of t
    |Jq_object of t
    |Jq_colon of t * t  (* to make an object *)
    |Jq_comma of t * t  (* to make an array *)
    |Jq_Ant of Loc.t * string
    |Jq_nil (* similiar to StNil *)
  let rec t_of_list lst = match lst with
    |[] -> Jq_nil
    | b::bs -> Jq_comma (b, t_of_list bs)
end

include Jq_ast

module MGram = MakeGram(Lexer)

let json  = MGram.Entry.mk "json"
let json_eoi = MGram.Entry.mk "json_eoi"


EXTEND  MGram
  GLOBAL: json_eoi  json ;
  json_eoi : [[ x = json ; EOI -> x ]];
```

```
  json :
    [[ "null" -> Jq_null
     |"true" -> Jq_bool true
     |"false" -> Jq_bool false

     | `ANTIQUOT (""|"bool"|"int"|"floo"|"str"|"list"|"alist" as n , s) ->
       Jq_Ant(_loc, n ^ ": " ^ s )

     | n = [ x = INT-> x | x =  FLOAT -> x ] -> Jq_number (float_of_string n)
     | "["; es = SELF ; "]" -> Jq_array es
     | "{";  kvs = SELF ;"}" -> Jq_object kvs

     | k= SELF; ":" ; v = SELF -> Jq_colon (k, v)
     | a = SELF; "," ; b = SELF -> Jq_comma (a, b)
     | -> Jq_nil  (* camlp4 parser epsilon has a lower priority *)

     ]];
END ;;

module AQ = Syntax.AntiquotSyntax
module Q = Syntax.Quotation
let destruct_aq s =
  let pos = String.index s ':' in
  let len = String.length s in
  let name = String.sub s 0 pos in
  let code = String.sub s (pos+1) (len-pos-1) in
  name, code

(** alternative*)
let destruct_aq2  = function (RE (_* Lazy as  name ) ":" (_* as content)) -> n
```

```
let /(_* Lazy as x) ":" (_* as rest ) / = "ghsoghos:ghsogh: ghsohgo";;
val rest : string = "ghsogh: ghsohgo"

val x : string = "ghsoghos"
```

```
let try /(_* Lazy as x) ":" (_* as rest ) / = "ghsoghosghsog ghsohgo"
in (x,rest)
with Match_failure _ -> ("","");;
```

notice that Syntax.AntiquotSyntax.(parse_expr,parse_patt) Syntax.(parse_implem, parse_interf)

```
        val parse_expr : Ast.loc -> string -> Ast.expr
        val parse_patt : Ast.loc -> string -> Ast.patt
    val parse_implem :
    val parse_interf :
```

```
let aq_expander = object
  inherit Ast.map as super
  method expr = function
    |Ast.ExAnt(_loc, s) ->
      let n, c = destruct_aq s in
      (** first round*)
      let e = AQ.parse_expr _loc c in
      begin match n with
        |"bool" -> <:expr< Jq_ast.Jq_bool $e$ >> (* interesting *)
        |"int" -> <:expr< Jq_ast.Jq_number (float $e$ ) >>
        |"flo" -> <:expr< Jq_ast.Jq_number $e$ >>
        |"str" -> <:expr< Jq_ast.Jq_string $e$ >>
        | "list" -> <:expr< Jq_ast.t_of_list $e$ >>
        |"alist" ->
          <:expr<
            Jq_ast.t_of_list
```

```
              (List.map (fun (k,v) -> Jq_ast.Jq_colon (Jq_ast.Jq_string k, v))
              $e$ )
            >>
          |_ -> e
        end
      |e -> super#expr e
    method patt = function
      | Ast.PaAnt(_loc,s) ->
        let n,c = destruct_aq s in
        AQ.parse_patt _loc c  (* ignore the tag *)
      | p -> super#patt p
  end
  module MetaExpr = struct
    (** the generator scans all the types defined in the current module
        then generate code for the last-appearing recursive bundle
    *)
    let meta_float' _loc f = <:expr< $`flo:f$ >>
    include Camlp4Filters.MetaGeneratorExpr(Jq_ast)
  end
  module MetaPatt = struct
    let meta_float' _loc f = <:patt< $`flo:f$ >>
    include Camlp4Filters.MetaGeneratorPatt(Jq_ast)
  end
  let (|>) x f = f x
  let parse_quot_string _loc s =
    let q = !Camlp4_config.antiquotations in
    (** checked by the lexer to allow antiquotation
        the flag is initially set to false, so antiquotations
        appearing outside a quotation won't be parsed
        *)
  Camlp4_config.antiquotations := true ;
```

```
let res = MGram.parse_string json_eoi _loc s in
 Camlp4_config.antiquotations := q ;
 res
let expand_expr _loc _ s =
  s
  |> parse_quot_string _loc
  |> MetaExpr.meta_t _loc
  |> aq_expander#expr
(* so it can appear in the toplevel *)
let expand_str_item _loc _ s =
  (**insert an expression as str_item *)
   <:str_item@_loc< $exp: expand_expr _loc None s $ >>
let expand_patt _loc _ s  =
  s
  |> parse_quot_string _loc
  |> MetaPatt.meta_t _loc
  |> aq_expander#patt
let _  =
  Q.add "json" Q.DynAst.expr_tag expand_expr ;
  Q.add "json" Q.DynAst.patt_tag expand_patt ;
  Q.add "json" Q.DynAst.str_item_tag expand_str_item ;
  Q.default := "json"
```

```
MGram.parse_string json_eoi Loc.ghost "[1,2]";;
 - : t = Jq_array (Jq_comma (Jq_number 1., Jq_number 2.))
MGram.parse_string json_eoi Loc.ghost "[1,2,]";;
- : t = Jq_array (Jq_comma (Jq_comma (Jq_number 1., Jq_number 2.), Jq_nil))
MGram.parse_string json_eoi Loc.ghost "1,2";;
- : t = Jq_comma (Jq_number 1., Jq_number 2.)
let alist = ["haha", <<1>>;"bob",<<3>>]  in <:json< [1 , $alist:alist$ ]>>;;
```

```
- : Json_anti.Jq_ast.t =
Json_anti.Jq_ast.Jq_array
 (Json_anti.Jq_ast.Jq_comma (Json_anti.Jq_ast.Jq_number 1.,
```

```
Json_anti.Jq_ast.Jq_comma
 (Json_anti.Jq_ast.Jq_colon (Json_anti.Jq_ast.Jq_string "haha",
   Json_anti.Jq_ast.Jq_number 1.),
 Json_anti.Jq_ast.Jq_comma
  (Json_anti.Jq_ast.Jq_colon (Json_anti.Jq_ast.Jq_string "bob",
    Json_anti.Jq_ast.Jq_number 3.),
  Json_anti.Jq_ast.Jq_nil))))
```

```
let b = << $ << 1 >>  $ >>  = << 1 >>;;
val b : bool = true
```

```
<< $ << 1 >> $>> --> parsing (my parser)
Jq_Ant(_loc, "<< 1 >> ") --> lifting  (mechnical)
Ex_Ant(_loc, "<< 1 >>") --> parsing  (the host parser )
<:expr< Jq_number 1. >>   --> antiquot_expand (my anti_expander )
<:expr < Jq_number 1. >>
*)
"json_anti.ml" : pp(camlp4of -filter meta)
<json_anti.{cmo,byte,native}> : pkg_dynlink, use_camlp4_full
```

4.7.9 Part 10 Lexer

Just follow the signature of module type Lexer is enough. generally you have to provide module Loc, Token, Filter, Error, and mk mk is essential

```
val mk : unit -> Loc.t -> char Stream.t -> (Token.t * Loc.t ) Stream.t
```

the verbose part lies in that you have to use the Camlp4.Sig.Loc, usually you have to maintain a mutable context, so when you lex a token, you can query the context to get Loc.t. you can refer Jake's jq_lexer.ml for more details. How about using lexer, parser all by myself? The work need to be done lies in you have to supply a plugin of type expand_fun, which is
`type 'a expand_fun = Ast.loc -> string option -> string -> 'a` so if you dont use ocamllexer, why bother the grammar module, just use lex yacc will make life easier, and you code will run faster .

```
type pos = {
  line : int;
  bol  : int;
  off  : int
};
type t = {
  file_name : string;
  start     : pos;
  stop      : pos;
  ghost     : bool
};
open Camlp4.PreCast
module Loc = Camlp4.PreCast.Loc
module Error : sig
  type t
  exception E of t
  val to_string : t -> string
  val print : Format.formatter -> t -> unit
end =  struct
  type t = string
  exception E of string
  let print = Format.pp_print_string (* weird, need flush *)
  let to_string  x  =  x
end
let _ =
  let module M = Camlp4.ErrorHandler.Register (Error) in ()
let (|> ) x  f = f x
module Token : sig
  module Loc : Camlp4.Sig.Loc
  type t
  val to_string : t -> string
```

```
  val print : Format.formatter -> t -> unit
  val match_keyword : string -> t -> bool
  val extract_string : t -> string
  module Filter : sig
    (* here t refers to the Token.t *)
    type token_filter = (t,Loc.t) Camlp4.Sig.stream_filter
    type t
    val mk : (string->bool)-> t
    val define_filter : t -> (token_filter -> token_filter) -> unit
    val filter : t -> token_filter
    val keyword_added : t -> string -> bool -> unit
    val keyword_removed : t -> string -> unit
  end
  module Error : Camlp4.Sig.Error
end = struct
  (** the token need not to be a variant with arms with KEYWORD
      EOI, etc, although conventional
  *)
  type t =
    | KEYWORD of string
    | NUMBER of string
    | STRING of string
    | ANTIQUOT of string * string
    | EOI
  let to_string t =
    let p = Printf.sprintf in
    match t with
      |KEYWORD s -> p "KEYWORD %S" s
      |NUMBER s -> p "NUMBER %S" s
      |STRING s -> p "STRING %S" s
      |ANTIQUOT (n,s) -> p "ANTIQUOT %S: %S" n s
```

```
    |EOI -> p "EOI"
let print fmt x = x |> to_string |> Format.pp_print_string fmt
let match_keyword kwd = function
  |KEYWORD k when kwd = k -> true
  |_ -> false

let extract_string = function
  |KEYWORD s | NUMBER s | STRING s -> s
  |tok -> invalid_arg ("can not extract a string from this token : "
                      ^ to_string tok)

module Loc = Camlp4.PreCast.Loc
module Error = Error
module Filter = struct
  type token_filter = (t * Loc.t ) Stream.t -> (t * Loc.t) Stream.t

  (** stub out *)
  (** interesting *)
  type t = unit

  (** the argument to mk is a function indicating whether
      a string should be treated as a keyword, and the default
      lexer uses it to filter the token stream to convert identifiers
      into keywords. if we want our parser to be extensible, we should
      take this into account
  *)
  let mk _ = ()
  let filter _ x  = x
  let define_filter _ _ = ()
  let keyword_added _ _ _ = ()
  let keyword_removed _ _ = ()
```

```
  end
end
module L = Ulexing
INCLUDE "/Users/bob/predefine_ulex.ml"
(* let rec token  c = lexer  *)
(*    | eof -> EOI  *)
(*    | newline -> token *)
(** TOKEN ERROR LOC
    mk : unit -> Loc.t -> char Stream.t -> (Token.t * Loc.t) Stream.t

    Loc.of_tuple :
    string * int * int * int * int * int * int * bool ->
    Loc.t
*)
```

4.8 Useful links

Abstract_Syntax_Tree
elehack
meta-guide
camlp4

Chapter 5

practical parts

5.1 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

5.1.1 Dev

- make changes in both .ml and .mli files

5.1.2 BOLT

5.2 Mikmatch

Directly supported in toplevel Regular expression *share* their own namespace.

1. compile

```
"test.ml" : pp(camlp4o -parser pa_mikmatch_pcre.cma)
<test.{cmo,byte,native}> : pkg_mikmatch_pcre
-- myocamlbuild.ml use default
```

2. toplevel

```
ocaml
#camlp4o ;;
#require "mikmatch_pcre" ;; (* make sure to follow the order strictly *)
```

3. debug

```
camlp4of -parser pa_mikmatch_pcre.cma -printer o test.ml
(* -no_comments does not work     *)
```

4. structure

   regular expressions can be used to match strings, it must be preceded by the RE keyword, or placed between slashes (/../).

```
match ... with pattern -> ...
function pattern -> ...
try ... with pattern -> ...
let /regexp/ = expr in expr
let try (rec) let-bindings in expr with pattern-match
(only handles exception raised by let-bindings)
MACRO-NAME regexp -> expr ((FILTER | SPLIT) regexp)
```

```
let x = (function (RE digit+) -> true | _ -> false) "13232";;
val x : bool = true
# let x = (function (RE digit+) -> true | _ -> false) "1323a2";;
val x : bool = true
# let x = (function (RE digit+) -> true | _ -> false) "x1323a2";;
val x : bool = false
```

```
let get_option () = match Sys.argv with
     [| _ |] -> None
    |[| _ ; RE (lower+ as key) "=" (_* as data) |] -> Some(key,data)
    |_ -> failwith "Usage: myprog [key=val]";;
val get_option : unit -> (string * string) option = <fun>
```

```
let option = try get_option () with Failure (RE "usage"~) -> None ;;
val option : (string * string) option = None
```

5. **sample regex** built in regexes

```
    lower, upper, alpha(lower|upper), digit, alnum, punct
    graph(alnum|punct), blank,cntrl,xdigit,space
    int,float
    bol(beginning of line)
    eol
    any(except newline)
    bos, eos
```

```
let f = (function (RE int as x : int) -> x ) "132";;
val f : int = 132
let f = (function (RE float as x : float) -> x ) "132.012";;
val f : float = 132.012
let f = (function (RE lower as x ) -> x ) "a";;
val f : string = "a"
let src = RE_PCRE int ;;
val src : string * 'a list = ("[+\\-]?(?:0(?:[Xx][0-9A-Fa-f]+|(?:[Oo][0-7]+|[Bb][01]+))|[0-9]+)", [])
let x = (function (RE _* bol "haha") -> true | _ -> false) "x\nhaha";;
val x : bool = true
```

```
RE hello = "Hello!"
RE octal  = ['0'-'7']
RE octal1 = ["01234567"]
RE octal2 = ['0' '1' '2' '3' '4' '5' '6' '7']
RE octal3 = ['0'-'4' '5'-'7']
RE octal4 = digit # ['8' '9']  (* digit is a predefined set of characters *)
RE octal5 = "0" | ['1'-'7']
```

```
RE octal6 = ['0'-'4'] | ['5'-'7']
RE not_octal = [ ^ '0'-'7'] (* this matches any character but an octal digit
RE not_octal' = [ ^ octal]  (* another way to write it *)

RE paren' = "(" _* Lazy ")"
(* _ is wild pattern, paren is built in *)
let p = function (RE (paren' as x )) -> x ;;
```

```
p "(xx))";;
- : string = "(xx)"
# p "(x)x))";;
- : string = "(x)"
```

```
RE anything  = _*         (* any string, as long as possible *)
RE anything' = _* Lazy    (* any string, as short as possible *)
RE opt_hello  = "hello"?       (* matches hello if possible, or nothing *)
RE opt_hello' = "hello"? Lazy (* matches nothing if possible, or hello *)
RE num = digit+         (* a non-empty sequence of digits, as long as possibl
                           shortcut for: digit digit* *)
RE lazy_junk = _+ Lazy (* match one character then match any sequence
                           of characters and give up as early as possible *)

RE at_least_one_digit = digit{1+}     (* same as digit+ *)
RE at_least_three_digits = digit{3+}
RE three_digits = digit{3}
RE three_to_five_digits = digit{3-5}
RE lazy_three_to_five_digits = digit{3-5} Lazy

let test s = match s with
    RE "hello" -> true
  | _ -> false
```

It's important to know that matching process will try *any* possible combination until the pattern is matched. However the combinations are tried from left to right, and repeats are either greedy or lazy. (greedy is default). laziness triggered by the presence of the Lazy keyword.

6. fancy features of regex

   (a) normal

```
let x = match "hello world" with
  RE "world" -> true
 | _ -> false;;
```

```
val x : bool = false
```

   (b) pattern match syntax (the let constructs can be used directly with a regexp pattern, but **let RE ... = ...** does not look nice, the sandwich notation (/.../) has been introduced)

```
Sys.ocaml_version;;
- : string = "3.12.1"
# RE num = digit + ;;
```

```
RE num = digit + ;;

let  /(num as major : int ) "." (num as minor : int)

( "." (num as patchlevel := fun s -> Some (int_of_string s))
| ("" as patchlevel := fun s -> None ))

( "+" (_*  as additional_info := fun s -> Some s )
| ("" as additional_info  := fun s -> None )) eos

/ = Sys.ocaml_version ;;
```

we always use **as** to extract the information.

```
val additional_info : string option = None
val major : int = 3
```

```
val minor : int = 12
val patchlevel : int option = Some 1
```

(c) File processing (Mikmatch.Text)

```
val iter_lines_of_channel : (string -> unit) -> in_channel -> unit
val iter_lines_of_file : (string -> unit) -> string -> unit
val lines_of_channel : in_channel -> string list
val lines_of_file : string -> string list
val channel_contents : in_channel -> string
val file_contents : ?bin:bool -> string -> string
val save : string -> string -> unit
val save_lines : string -> string list -> unit
exception Skip
val map : ('a -> 'b) -> 'a list -> 'b list
val rev_map : ('a -> 'b) -> 'a list -> 'b list
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
val map_lines_of_channel : (string -> 'a) -> in_channel -> 'a list
val map_lines_of_file : (string -> 'a) -> string -> 'a list
```

(d) **Mikmatch.Glob** (pretty useful)

```
val scan :
  ?absolute:bool ->
  ?path:bool ->
  ?root:string ->
  ?nofollow:bool -> (string -> unit) -> (string -> bool) list -> unit
val lscan :
  ?rev:bool ->
  ?absolute:bool ->
  ?path:bool ->
  ?root:string list ->
  ?nofollow:bool ->
```

```
    (string list -> unit) -> (string -> bool) list -> unit
  val list :
    ?absolute:bool ->
    ?path:bool ->
    ?root:string ->
    ?nofollow:bool -> ?sort:bool -> (string -> bool) list -> string list
  val llist :
    ?rev:bool ->
    ?absolute:bool ->
    ?path:bool ->
    ?root:string list ->
    ?nofollow:bool ->
    ?sort:bool -> (string -> bool) list -> string list list
```

here we want to get `~/.*/*.conf` file X.list (predicates corresponding to each layer .

```
let xs = let module X = Mikmatch.Glob in X.list ~root:"/Users/bob" [FILTER "." ; FILTER _* ".conf" eos ] ;;
val xs : string list = [".libfetion/libfetion.conf"]
```

```
let xs =
  let module X = Mikmatch.Glob in
  X.list ~root:"/Users/bob" [const true; FILTER _* ".pdf" eos ]
  in print_int (List.length xs) ;;
```

455

(e) Lazy or Greedy

```
match "acbde (result), blabla... " with
RE _* "(" (_* as x) ")" -> print_endline x | _ -> print_endline "Failed";;
```

result

```
 match "acbde (result),(bla)bla... " with
 RE _* Lazy "(" (_* as x) ")" -> print_endline x | _ -> print_endline "Fai
```

result),(bla

```
let / "a"? ("b" | "abc" ) as x / = "abc" ;; (* or patterns, the same as before*)
val x : string = "ab"
# let / "a"? Lazy ("b" | "abc" ) as x / = "abc" ;;
val x : string = "abc"
```

In place conversions of the substrings can be performed, using either the predefined converters *int*, *float*, or custom converters

```
let z  = match "123/456" with RE (digit+ as x : int ) "/" (digit+ as y : int) -> x ,y ;;
val z : int * int = (123, 456)
```

Mixed pattern

```
let z = match 123,45, "6789" with i,_, (RE digit+ as j : int) | j,i,_ -> i * j + 1;;
val z : int = 835048
```

(f) Backreferences

Previously matched substrings can be matched again using backreferences.

```
let z =  match "abcabc" with RE _* as x !x -> x ;;
val z : string = "abc"
```

(g) Possessiveness prevent backtracking

```
let x = match "abc" with RE _* Possessive _ -> true | _ -> false;;
val x : bool = false
```

(h) macros

i. FILTER macro

```
let f = FILTER int eos;;
val f : ?share:bool -> ?pos:int -> string -> bool = <fun>
# f "32";;
- : bool = true
# f "32a";;
- : bool = false
```

ii. REPLACE macro

```
let remove_comments = REPLACE "#" _* Lazy eol -> "" ;;
val remove_comments : ?pos:int -> string -> string = <fun>
# remove_comments "Hello #comment \n world #another comment" ;;
```

```
- : string = "Hello \n world "
let x = (REPLACE "," -> ";;" ) "a,b,c";;
val x : string = "a;;b;;c"
```

iii. REPLACE_FIRST macro

iv. SEARCH(_FIRST) COLLECT COLLECTOBJ MACRO

```
let search_float = SEARCH_FIRST float as x : float -> x ;;
val search_float : ?share:bool -> ?pos:int -> string -> float = <fun>
search_float "bla bla -1.234e12 bla";;
- : float = -1.234e+12
let get_numbers = COLLECT float as x : float -> x ;;
val get_numbers : ?pos:int -> string -> float list = <fun>
get_numbers "1.2   83  nan  -inf 5e-10";;
- : float list = [1.2; 83.; nan; neg_infinity; 5e-10]
let read_file = Mikmatch.Text.map_lines_of_file (COLLECT float as x : float -> x );;
val read_file : string -> float list list = <fun>

(** Negative assertions *)
let get_only_numbers =  COLLECT < Not alnum . > (float as x : float) < . Not alnum > -> x

let list_words = COLLECT (upper | lower)+ as x -> x ;;
val list_words : ?pos:int -> string -> string list = <fun>
# list_words "gshogh sghos sgho ";;
- : string list = ["gshogh"; "sghos"; "sgho"]
RE pair = "(" space* (digit+ as x : int) space* ","  space* ( digit + as y : int ) space* ")";;
 # let get_objlist = COLLECTOBJ pair;;
val get_objlist : ?pos:int -> string -> < x : int; y : int > list =
```

v. SPLIT macro

```
let ys = (SPLIT space* [",;"] space* ) "a,b,c, d, zz;";;
val ys : string list = ["a"; "b"; "c"; "d"; "zz"]
let f = SPLIT space* [",;"] space* ;;
val f : ?full:bool -> ?pos:int -> string -> string list = <fun>
```

Full is false by default. When true, it considers the regexp as a separator between substrings even if the first or the last one is empty. will add some whitespace trailins

```
f ~full:true "a,b,c,d;"  ;;
- : string list = ["a"; "b"; "c"; "d"; ""]
```

vi. MAP macro (a weak lexer) (MAP regexp -> expr)

splits the given string into fragments: the fragments that do not match the pattern are returned as *'Text s*. Fragments that match the pattern are replaced by the result of expr

```
let f = MAP ( "+" as x = 'Plus ) -> x ;;
val f : ?pos:int -> ?full:bool -> string -> [> 'Plus | 'Text of string ] list =
let x =  (MAP ',' -> 'Sep ) "a,b,c";;
val x : [> 'Sep | 'Text of string ] list = ['Text "a"; 'Sep; 'Text "b"; 'Sep; 'Text "c"]
```

```
let f = MAP ( "+" as x = 'Plus ) | ("-" as x = 'Minus) | ("/" as x = '
  | ("*" as x = 'Mul) | (digit+ as x := fun s -> 'Int (int_of_string s
  | (alpha [alpha digit] +  as x := fun s -> 'Ident s) -> x ;;
val f :
  ?pos:int ->
  ?full:bool ->
  string ->
  [> 'Div
   | 'Ident of string
   | 'Int of int
   | 'Minus
   | 'Mul
   | 'Plus
   | 'Text of string ]
list = <fun>
# f "+-*/";;
- : [> 'Div
     | 'Ident of string
     | 'Int of int
     | 'Minus
     | 'Mul
     | 'Plus
     | 'Text of string ]
    list
=
```

```
[`Text ""; `Plus; `Text ""; `Minus; `Text ""; `Mul; `Text ""; `Div; `T
let xs = Mikmatch.Text.map (function `Text (RE space* eos) -> raise Mi
val xs :
  [> `Div
   | `Ident of string
   | `Int of int
   | `Minus
   | `Mul
   | `Plus
   | `Text of string ]
  list = [`Plus; `Minus; `Mul; `Div]
```

vii. lexer (ulex is faster and more elegant)

```
let get_tokens = f |- Mikmatch.Text.map (function `Text (RE space* eos
-> raise Mikmatch.Text.Skip | `Text x -> invalid_arg x | x
-> x) ;;

val get_tokens :
  string ->
  [> `Div
   | `Ident of string
   | `Int of int
   | `Minus
   | `Mul
   | `Plus
   | `Text of string ]
  list = <fun>

get_tokens "a1+b3/45";;
- : [> `Div
     | `Ident of string
     | `Int of int
```

```
          | `Minus
          | `Mul
          | `Plus
          | `Text of string ]
        list
    = [`Ident "a1"; `Plus; `Ident "b3"; `Div; `Int 45]
```

viii. SEARCH macro (location)

```
let locate_arrows = SEARCH %pos1 "->" %pos2 ->  Printf.printf "(%i-%i)" pos1 (pos2-1);;
val locate_arrows : ?pos:int -> string -> unit = <fun>
# locate_arrows "gshogho->ghso";;
(7-8)- : unit = ()
let locate_tags = SEARCH "<" "/"? %tag_start (_* Lazy as tag_contents) %tag_end ">" -> Printf.printf "
```

(i) debug

```
let src = RE_PCRE <Not alnum . > (float as x : float ) < . Not alnum > in print_endline (fst src);;
(?<![0-9A-Za-z])([+\-]?(?:(?:[0-9]+(?:\.[0-9]*)?|\.[0-9]+)(?:[Ee][+\-]?[0-9]+)?|(?:[Nn][Aa][Nn]|[Ii][Nn][Ff
```

(j) ignore the case

```
match "OCaml" with RE "O" "caml"~ -> print_endline "success";;
success
```

(k) zero-width assertions

```
RE word =  < Not alpha . >    alpha+ < . Not alpha>
RE word' = < Not alpha . >    alpha+ < Not alpha >
```

```
RE triplet = <alpha{3} as x>
let print_triplets_of_letters = SEARCH triplet -> print_endline x
print_triplets_of_letters "helhgoshogho";;
```

```
hel
elh
lhg
hgo
gos
osh
```

```
sho
hog
ogh
gho
- : unit = ()
```

```
(SEARCH alpha{3} as x -> print_endline x ) "hello world";;
```

```
hel
wor
```

```
(SEARCH <alpha{3} as x> -> print_endline x ) "hello world";;
```

```
hel
ell
llo
wor
orl
rld
```

```
(SEARCH alpha{3} as x -> print_endline x ) ~pos:2 "hello world";;
```

```
llo
wor
```

(l) dynamic regexp

```
let get_fild x = SEARCH_FIRST @x "=" (alnum* as y) -> y;;
val get_fild : string -> ?share:bool -> ?pos:int -> string -> string = <fun>
# get_fild "age" "age=29 ghos";;
- : string = "29"
```

(m) reuse

using macro INCLUDE

(n) view patterns

```
let view XY = fun obj -> try Some (obj#x, obj#y) with _ -> None ;;
val view_XY : < x : 'a; y : 'b; .. > -> ('a * 'b) option = <fun>
# let test_orign = function
```

```
   %XY (0,0) :: _ -> true
  |_ -> false
;;
      val test_orign : < x : int; y : int; .. > list -> bool = <fun>


let view Positive = fun x -> x > 0
let view Negative = fun x -> x <= 0

let test_positive_coords = function
  %XY ( %Positive, %Positive ) -> true
  | _ -> false

  (** lazy pattern is already supported in OCaml *)
let test x = match x with
    lazy v -> v

type 'a lazy_list = Empty | Cons of ('a * 'a lazy_list lazy_t)


let f = fun (Cons (_ , lazy (Cons (_, lazy (Empty)) ) )) -> true ;;
let f = fun %Cons (x1, %Cons (x2 %Empty)) -> true  (* simpler *)
```

implementation let view X = f is translated into: let view_X = f

Similarly, we have local views: let view X = f in ...

Given the nature of camlp4, this is the simplest solution that allows us to make views available to other modules, since they are just functions, with a standard name. When a view X is encountered in a pattern, it uses the view_X function. The compiler will complain if doesn't have the right type, but not the preprocessor.

About inline views: since views are simple functions, we could insert functions directly in patterns. I believe it would make the pattern really difficult to read, especially since views are expected to be most useful in already complex patterns.

About completeness checking: our definition of views doesn't allow the compiler to warn against incomplete or redundants pattern-matching. We have the same situation with regexps. What we define here are incomplete or overlapping views, which have a broader spectrum of applications than views which are defined as sum types.

(o) tiny use

```
se (FILTER _* "map_lines_of_file" ) "Mikmatch";;
val map_lines_of_file : (string -> 'a) -> string -> 'a list
```

```
let _  = Mikmatch.map_lines_of_file
  (function x ->
    match x with
      | RE "\xbegin{bluecode}" -> "\n" ^ x
      | RE "\xend{bluecode}" -> x ^ ``\n''
      | _  -> x  )
  "/Users/bob/SourceCode/Notes/ocaml-hacker.tex"
  |> List.enum
  |> File.write_lines "/Users/bob/SourceCode/Notes/ocaml-hacker-back-up.te
```

5.3 pa-do

5.4 num

- delimited overloading

5.5 caml-inspect

It's mainly used to debug programs or presentation. blog

1. usage

```
#require "inspect";;
open Inspect ;;

Sexpr.(dump (test_data ()))
Sexpr.(dump dump) (** can dump any value, including closure *)
Dot.(dump_osx dump_osx)
```

2. *module Dot*

```
    dump
    dump_to_file
    dump_with_formatter
    dump_osx
```

3. *module Sexpr*

```
    dump
    dump_to_file
    dump_with_formatter
```

4. principle

   OCaml values all share a *common low-level* representation. The basic building block that is used by the runtime-system(which is written in the C programming languag) to represent any value in the OCaml universe is the value type. Values are always *word-sized*. A word is either 32 or 64 bits wide(*Sys.word_size*)

   A value can either be a pointer to a block of values in the OCaml heap, a pointer to an object outside of the heap, or an unboxed integer. Naturally, blocks in the heap are garbage-collected.

   To distinguish between unboxed integers and pointers, the system uses the least-significant bit of the value as a flag. If the LSB is set, the value is unboxed. If the LSB is cleared, the value is a pointer to some other region of memory. This encoding also explains why the int type in OCaml is only 31 bits wide (63 bits wide on 64 bit platforms).

Since blocks in the heap are garbage-collected, they have strict structure constraints. Information like the tag of a block and its size(in words) is encoded in the header of each block.

There are two categories of blocks with respect to the garbage collector:

(a) Structured blocks
May only contain well-formed values, as they are recursively traversed by the garbage collector.

(b) Raw blocks
are not scanned by the garbage collector, and can thus contain arbitrary values.

Structured blocks have tag values lower than *Obj.no_scan_tag*, while raw blocks have tags equal or greater than *Obj.no_scan_tag*.

The type of a block is its tag, which is stored in the block header.(*Obj.tag*)

```
Obj.(let f ()= repr |- tag in no_scan_tag, f () 0, f () [|1.;2.|], f
() (1,2) ,f ()[|1,2|]);;

- : int * int * int * int * int = (251, 1000, 254, 0, 0)

se_str "_tag" "Obj";;

    external tag : t -> int = "caml_obj_tag"
    external set_tag : t -> int -> unit = "caml_obj_set_tag"
    val lazy_tag : int
    val closure_tag : int
    val object_tag : int
    val infix_tag : int
    val forward_tag : int
    val no_scan_tag : int
    val abstract_tag : int
    val string_tag : int
```

```
val double_tag : int
val double_array_tag : int
val custom_tag : int
val final_tag : int
val int_tag : int
val out_of_heap_tag : int
val unaligned_tag : int
```

(a) *0 to Obj.no_scan_tag-1* A structured block (an array of Caml objects). Each field is a value.

(b) *Obj.closure_tag*: A closure representing a functional value. The first word is a pointer to a piece of code, the remaining words are values containing the environment.

(c) *Obj.string_tag*: A character string.

(d) *Obj.double_tag*: A double-precision floating-point number.

(e) *Obj.double_array_tag*: An array or record of double-precision floating-point numbers.

(f) *Obj.abstract_tag*: A block representing an abstract datatype.

(g) *Obj.custom_tag*: A block representing an abstract datatype with user-defined finalization, comparison, hashing, serialization and deserialization functions attached

(h) *Obj.object_tag*: A structured block representing an object. The first field is a value that describes the class of the object. The second field is a unique object id (see *Oo.id*). The rest of the block represents the variables of the object.

(i) *Obj.lazy_tag, Obj.forward_tag*: These two block types are used by the runtime-system to implement lazy-evaluation.

(j) *Obj.infix_tag*: A special block contained within a closure block

5. representation

For atomic types

(a) int, char (ascii code) : Unboxed integer values

(b) float : Blocks with tag *Obj.dobule_tag*

(c) string : Blocks with tag *Obj.string_tag*

(d) int32, int64, nativeint : Blocks with *Obj.custom_tag*

For Tuples and records: Blocks with tag 0

```
Obj.((1,2) |> repr |> tag);;
- : int = 0
```

For normal array(except float array), Blocks with tag 0

For Arrays and records of floats: Block with tag *Obj.double_array_tag*

For concrete types,

(a) Constant ctor : Represented by unboxed integers(0,1,...).

(b) Non-Constant ctor: Block with a tag lower than *Obj.no_scan_tag* that encodes the constructor, numbered in order of declaration, starting at 0.

For objects: Blocks with tag *Obj.object_tag*. The first field refers to the class of the object and its associated method suite. The second field contains a unique object ID. The remaining fields are the instance variables of the object.

For polymorphic variants: Variants are similar to constructed terms. There are a few differences

(a) Variant constructors are identified by their hash value

(b) Non-constant variant constructors are not flattened. They are always block of size 2, where the first field is the hash. The second field can either contain a single value or a pointer to another structured block(just like a tuple)

5.6 ocamlgraph

ocamlgraph is a sex library which deserve well-documentation.

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

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

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

   Follow this file, you could know how to build a graph, A nice trick, to bind open command to use graphviz to open the file, then it will do the sync automatically and you can *#u "open *.dot"*, so nice

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

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

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

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

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

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

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

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

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

```
    f g_intersect "g_intersect.dot";
    f g_union "g_union.dot"
    end
  )
```

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

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

Different modules have corresponding algorithms

2. hierachical

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

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

Graph.Pack requires its label being integer

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

```
    module Digraph :
        module V :
        module E :
```

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

3. hierachical for undirected graph

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


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

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

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

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

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

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

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

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

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

```
module V = struct
  type t = string
  let compare = Pervasives.compare
  let hash = Hashtbl.hash
  let equal = (=)
end
```

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

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

    Not only that, but the V and E structure will work for
    persistent and directed graphs that are concretelabeled,
    and you can switch by replacing Imperative with Persistent
    , and Graph with Digraph.
    *)

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

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

4. another example (edge unlabeled, directed graph)

```
open Graph
module V = struct
  type t = string
  let compare = Pervasives.compare
  let hash = Hashtbl.hash
  let equal = (=)
end
module G = Imperative.Digraph.Concrete (V)
let g = G.create ()
```

```
let _ = G.(begin
  add_edge g "a" "b";
  add_edge g "a" "c";
  add_edge g "b" "d";
  add_edge g "b" "d"
end )
module Display = struct
  include G
  let vertex_name v = (V.label v)
  let graph_attributes _ = []
  let default_vertex_attributes _ = []
  let vertex_attributes _ = []
  let default_edge_attributes _ = []
  let edge_attributes _ = []
  let get_subgraph _ = None
end
module Dot_ = Graphviz.Dot(Display)
let _ =
  let out = open_out "g.dot" in
  finally (fun _ -> close_out out) (fun g ->
    let fmt =
      (out |> Format.formatter_of_output) in
    Dot_.fprint_graph fmt g ) g
```

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

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

```
open Batteries_uni
open Graph
module V = struct
  type t = string
```

```
  let compare = Pervasives.compare
  let hash = Hashtbl.hash
  let equal = (=)
end
module StringDigraph = Imperative.Digraph.Concrete (V)
module Display = struct
  include StringDigraph
  open StringDigraph
  let vertex_name v = (V.label v)
  let graph_attributes _ = []
  let default_vertex_attributes _ = []
  let vertex_attributes _ = []
  let default_edge_attributes _ = []
  let edge_attributes _ = []
  let get_subgraph _ = None
end

module DisplayG = Graphviz.Dot(Display)


let dot_output g file =
  let out = open_out file in
  finally (fun _ -> close_out out) (fun g ->
    let fmt =
      (out |> Format.formatter_of_output) in
    DisplayG.fprint_graph fmt g ) g


let g_of_edges edges = StringDigraph.(
  let g = create () in
  let _ = Stream.iter (fun (a,b) -> add_edge g a  b) edges in
```

```
  g
)

let line = "path.ml: Hashtbl Heap List Queue Sig Util"

let edges_of_line line =
  try
    let (a::b::res) =
      Pcre.split ~pat:".ml:" ~max:3  line in
    let v_a =
      let _ =  a.[0]<- Char.uppercase a.[0] in
      a  in
    let v_bs =
      (Pcre.split ~pat:"\\s+" b ) |> List.filter (fun x -> x <> "") in
    let edges = List.map (fun v_b -> v_b, v_a ) v_bs in
    edges
  with exn -> invalid_arg ("edges_of_line : " ^ line)

let lines_stream_of_channel chan = Stream.from (fun _ ->
    try Some (input_line chan) with End_of_file -> None );;

let edges_of_channel chan = Stream.(
  let lines = lines_stream_of_channel chan in
  let edges = lines |> map (edges_of_line |- of_list) |> concat in
  edges
)


let graph_of_channel = edges_of_channel |- g_of_edges

let _ =
```

```
let stdin = open_in Sys.argv.(1) in
let g = graph_of_channel stdin in begin
Printf.printf "writing to dump.dot\n";
dot_output g "dump.dot";
Printf.printf "finished\n"
end
```

5.7 pa-monad

1. debug
   tags file

```
  "monad_test.ml" : pp(camlp4o -parser pa_monad.cmo)
  camlp4o -parser pa_monad.cmo monad_test.ml -printer o

  (** filter *)

  let a = perform let b = 3 in  b
  let bind x f = f x
  let c = perform c <-- 3 ; c
  (* output
  let a = let b = 3 in b
  let bind x f = f x
  let c = bind 3 (fun c -> c)
  *)



let bind x f = List.concat (List.map f x)
let return x = [x]
let bind2 x f = List.concat (List.map f x)

let c = perform
    x <-- [1;2;3;4];
    y <-- [3;4;4;5];
    return (x+y)


let d = perform with bind2 in
    x <-- [1;2;3;4];
    y <-- [3;4;4;5];
    return (x+y)
```

```
let _ = List.iter print_int c
let _ = List.iter print_int d

(*
let bind x f = List.concat (List.map f x)
let return x = [ x ]
let bind2 x f = List.concat (List.map f x)
let c =
  bind [ 1; 2; 3; 4 ]
    (fun x -> bind [ 3; 4; 4; 5 ] (fun y -> return (x + y)))
let d =
  bind2 [ 1; 2; 3; 4 ]
    (fun x -> bind2 [ 3; 4; 4; 5 ] (fun y -> return (x + y)))
let _ = List.iter print_int c
let _ = List.iter print_int d
*)
```

2. translation rule

   it's simple. **perform** or **perform with bind in** then it will translate all phrases ending with *;; x <– me;* will be translated into *me »= (fun x ->);* *me;* will be translated into *me »= (fun _ -> ...)* you should refer *pa_monad.ml* for more details *perform with exp1 and exp2 in exp3* uses the first given expression as bind and the second as match-failure function. *perform with module Mod in exp* use the function named bind from module Mod. In addition ues the module's failwith in refutable patterns

```
let a = perform with (flip Option.bind) in a <-- Some 3;  b<-- Some 32; Some (a+ b) ;;
val a : int option = Some 35
```

   it will be translated into

```
let a =
  flip Option.bind (Some 3)
    (fun a -> flip Option.bind (Some 32) (fun b -> Some (a + b))
      )
```

3. ParameterizedMonad

```
class ParameterizedMonad m where
  return :: a -> m s s a
  (>>=) :: m s1 s2 t -> (t -> m s2 s3 a) -> m s1 s3 a

data Writer cat s1 s2 a = Writer {runWriter :: (a, cat s1 s2)}

instance (Category cat) => ParameterizedMonad (Writer cat) where
  return a = Writer (a,id)
  m >>= k = Writer $ let
    (a,w) = runWriter
    (b,w') = runWriter (k a)
    in (b, w' . w)
```

```
module State : sig
  type ('a,'s) t = 's -> ('a * 's)
  val return : 'a -> ('a,'s) t
  val bind : ('a,'s ) t -> ('a -> ('b,'s) t ) -> ('b,'s) t
  val put : 's -> (unit,'s) t
  val get :  ('s,'s) t
end = struct
 type ('a,'s) t = ('s -> ('a * 's))
 let return v = fun s -> (v,s)
 let bind (v : ('a,'s) t) (f : 'a -> ('b,'s) t) : ('b,'s) t =
    fun s ->
   let a,s' = v s in
   let a',s'' = f a s' in
   (a',s'')
 let put s = fun _ -> (), s
 let get = fun s -> s,s
end


module PState : sig
  type ('a, 'b, 'c) t = 'b -> 'a * 'c
  val return : 'a -> ('a,'b,'b) t
  val bind : ('b,'a,'c)t -> ('b -> ('d,'c, 'e) t ) -> ('d,'a,'e)
      t
  val put : 's -> (unit,'b,'s)t
  val get : ('s,'s,'s) t
end  = struct
```

```
 type ('a,'s1,'s2) t = 's1 -> ('a * 's2)
 let return v = fun s -> (v,s)
 let bind v f = fun s ->
   let a,s' = v s in
   let a',s'' = f a s' in
   (a',s'')
 let put s = fun _ -> (), s
 let get = fun s -> s,s
end
```

```
let v = State.(perform  x <-- return 1 ; y <-- return 2 ; let _ =
print_int (x+y) in return (x+y) );;
```

```
val v : (int, '_a) State.t = <fun>
```

```
let v = State.(perform x <-- return 1 ; y <-- return 2 ; z <-- get ; put (x+
  z<-- get ; let _ = print_int z in return (x+y+z));;
```

```
 val v : (int, int) State.t = <fun>
```

```
  v 3;;
6- : int * int = (9, 6)
```

```
let v = PState.(perform x <-- return 1 ; y <-- return 2 ; z <-- get ; put (x
z<-- get ; let _ = print_int z in return (x+y+z));;
```

```
val v : (int, int, int) PState.t = <fun>
```

```
v 3 ;;
6- : int * int = (9, 6)
```

```
let v = PState.(perform x <-- return 1 ; y <-- return 2 ; z <-- get ;
put (string_of_int (x+y+z)) ; return z );;
```

```
val v : (int, int, string) PState.t = <fun>
```

```
# v 3;;
v 3;;
- : int * string = (3, "6")
```

5.8 bigarray

This implementation allows efficient sharing of large numerical arrays between Caml code and C or Fortran numerical libraries. Your are encouraged to

```
open Bigarray
```

. Big arrays support the ad-hoc polymorphic operations (comparison, hashing,marshall)

Element kinds

The abstract type

captures type 'a for values read or written in the array, while 'b which represents the actual content of the big array.

Array layouts

5.9 sexplib

Basic Usage

```
#require "sexplib.top";;
```

```
open Sexplib
open Std
type t = A of int list | B with sexp;;
module S = Sexp;;
module C = Conv;;
```

```
 sub_modules "Sexplib";;
module This_module_name_should_not_be_used :
    module Type :
    module Parser :
    module Lexer :
    module Pre_sexp :
        module Annot :
```

```
        module Parse_pos :
        module Annotated :
        module Of_string_conv_exn :
    module Sexp_intf :
        module type S =
            module Parse_pos :
            module Annotated :
            module Of_string_conv_exn :
    module Sexp :
        module Parse_pos :
        module Annotated :
        module Of_string_conv_exn :
    module Path :
    module Conv :
        module Exn_converter :
    module Conv_error :
    module Exn_magic :
    module Std :
        module Hashtbl :
            module type HashedType =
            module type S =
            module Make :
        module Big_int :
        module Nat :
        module Num :
        module Ratio :
        module Lazy :
```

Build

build with sexplib

Modules

`Sexp` Contains all I/O-functions for Sexp, module `Conv` helper functions converting OCaml-valus of `standard-types` to Sexp. Moduel `Path` supports sub-expression extraction and substitution.

Sexp

```
type t = Sexplib.Type.t = Atom of string | List of t list
```

Syntax

`with sexp` or `with sexp_of` or `with of_sexp`. `signatures` are also well supported. When packed, you should use `TYPE_CONV_PATH` to make the location right. Common utilities are exported by `Std`.

we hope `sexp_of_t |- t_of_sexp` to be an `id` function

```
let f = exp_of_int |- int_of_exp
Enum.( let a = range ~until:max_int min_int in
       fold2 (fun l r a -> a & (l=r)) true a (map f a) );;
```

5.10 bin-prot

5.11 fieldslib

5.12 variantslib

5.13 delimited continuations

Continuatioins A conditional banch selects a continuation from the two possible futures; rasing an exception discards. Traditional way to handle continuations explicitly in a program is to transform a program into cps style. Continuation captured by call/cc is the **whole** continuation that includes all the future computation.. In practice, most of the continuations that we want to manipulate are only a part of computation. Such continuations are called **delimited continuations** or **partial continuations**.

1. cps transform

   there are multiple ways to do cps transform, here are two.

```
------------------------------------------
   [x] --> x
   [\x. M] --> \k . k (\x . [M])
   [M N] --> \k. [M] (\m . m [N] k)
------------------------------------------



------------------------------------------
   [x] --> \k . k x
   [\x. M] --> \k. k (\x.[M])
   [M N] --> \k. [M] (\m . [N] (\n. m n k))
------------------------------------------


[callcc (\k. body)] = \outk. (\k. [body] outk) (\v localk. outk
   v)
```

2. experiment

```
#load "delimcc.cma";;
```

```
Delimcc.shift;;
- : 'a Delimcc.prompt -> (('b -> 'a) -> 'a) -> 'b = <fun>
```

```
reset (fun () -> M ) --> push_prompt p (fun () -> M )
shift (fun k -> M) --> shift p (fun k -> M )
```

in racket you should have *(require racket/control)* and then *(reset expr ...+)* *(shift id expr ...+)*

```
module D = Delimcc
(** set the prompt *)
let p = D.new_prompt ()
let (reset,shift),abort  = D.(push_prompt &&& shift &&& abort ) p;;
let foo x = reset (fun () -> shift (fun cont -> if x = 1 then cont 10 else 20
```

```
foo 1 ;;
- : int = 110
foo 2  ;;
- : int = 20
5 * reset (fun () -> shift (fun k -> 2 * 3 ) + 3 * 4 );;
- : int = 30
reset (fun () -> 3 + shift (fun k -> 5 * 2) ) - 1 ;;
- : int = 9
```

```
val p : '_a D.prompt = <abstr>
val reset : (unit -> '_a) -> '_a = <fun>
val shift : (('_a -> '_b) -> '_b) -> '_a = <fun>
val abort : '_a -> 'b = <fun>
```

```
let p = D.new_prompt ()
let (reset,shift),abort  = D.( push_prompt &&& shift &&& abort ) p;;
```

```
reset (fun () -> if (shift (fun k -> k(2 = 3))) then "hello" else "hi ") ^ "world";;
- : string = "hi world"
reset (fun () -> if (shift (fun k -> "laji")) then "hello" else "hi ") ^ "world";;
- : string = "lajiworld"
reset (fun _ -> "hah");;
- : string = "hah"
```

```
let make_operator () =
  let p = D.new_prompt () in
  let (reset,shift),abort = D.( push_prompt &&& shift &&& abort) p in
  p,reset,shift,abort
```

Delimited continuations seems not able to handle answer type polymorphism.

```
exception Str of [`Found of int | `NotFound]
```

```
let times lst  =
  let rec times_aux lst = match lst with
```

```
    | [] -> 1
    | 0 :: xs -> shift (fun _ -> 0 )
    | x :: xs -> begin
      (* printf "entering %d\n" x ; *)
      let v = x * times_aux xs in
      (* printf "exiting %d\n" x ;  *)
      v
    end in
  reset (fun () -> times_aux lst )
```

Store the continuation, the type system is not friendly to the continutations, but fortunately we have *side effects* at hand, we can store it. (This is pretty hard in Haskell)

```
let p,reset,shift,abort = make_operator() in
  let c = ref None in
  begin
   reset (fun () -> 3 + shift (fun k -> c:= Some k ;  0) - 1)  ;
   Option.get (!c) 20
   end ;;
          Characters 81-139:
     reset (fun () -> 3 + shift (fun k -> c:= Some k ;  0) - 1)  ;
     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
     Warning 10: this expression should have type unit.

- : int = 22

let cont =
  let p,reset,shift,abort = make_operator() in
  let c = ref None in
  let rec id lst = match lst with
    | [] -> shift (fun k -> c:=Some k ; [] )
    |x :: xs -> x :: id xs in
```

```
  let xs = reset (fun () -> id [1;2;3;4]) in
  xs, Option.get (!c);;

val cont : int list * (int list -> int list) = ([], <fun>)
```

```
# let a,b = cont ;;
val a : int list = []

val b : int list -> int list = <fun>
# b [];;
- : int list = [1; 2; 3; 4]
```

```
type tree = Empty | Node of  tree * int  * tree
let walk_tree =
  let cont = ref None in
  let p,reset,shift,abort = make_operator() in
  let yield n = shift (fun k -> cont := Some k; print_int n ) in
  let rec walk2 tree = match tree with
    |Empty -> ()
    |Node (l,v,r) ->
      walk2 l ;
      yield v ;
      walk2 r in
  fun tree -> (reset (fun _ -> walk2 tree ), cont);;

val walk_tree : tree_t -> unit * ('_a -> unit) option Batteries.ref =
```

```
# let _, cont = walk_tree tree1 ;;
1val cont : ('_a -> unit) option Batteries.ref = {contents = Some <fun>}
# Option.get !cont ();;
2- : unit = ()
# Option.get !cont ();;
3- : unit = ()
# Option.get !cont ();;
- : unit = ()
# Option.get !cont ();;
- : unit = ()
```

It's quite straightforward to implement yield using delimited continuation, since each time shifting will escape the control, and you store the continuation, later it can be resumed.

```
(** defer the continuation *)
shift (fun k -> fun () -> k "hello")
```

By wrapping continuations, we can **access the information outside** of the enclosing reset while staying within reset lexically.

suppose this type check

```
let f x = reset (fun () -> shift (fun k -> fun () -> k "hello") ^ "world" ) x
f : unit -> string
```

3. Answer type modification (serious) in the following context, `reset (fun () -> [...] ^ "wo` the value returned by reset appears to be a string. An answer type is a type of the enclosing *reset*.

4. reorder delimited continuations

   if we apply a continuation at the tail position, the captured computation is simply resumed. If we apply a continuation at the non-tail position, we can perform additional computation after resumed computation finishes.

   Put differently, we can switch the execution order of the surrounding context.

```
let p,reset,shift,abort = make_operator () in
    reset (fun () -> 1 + (shift (fun k -> 2 * k 3 )));;
```

```
- : int = 8
```

```
let p,reset,shift,abort = make_operator () in
   let either a b = shift (fun k -> k a ; k b ) in
   reset (fun () ->
   let x = either  0  1 in
   print_int x ; print_newline ());;
```

```
0
1
```

5. useful links

   sea side

   shift and reset tutorial

   shift reset tutorial

   racket control operators

   caml-shift-paper.pdf

   caml-shift-talk

5.14 shcaml

A shell library. (you can refer `Shell` module of shell package)

All modules in the system are submodules of the `Shcaml` module, except ofr the module `Shtop`

5.15 deriving

Build

For debuging

```
cd `camlp4 -where`
ln -s `ocamlfind query deriving-ocsigen`/pa_deriving.cma
```

So you could type `camlp4o -parser pa_deriving.cma test.ml`

Toplevel `#require "deriving-ocsigen.syntax";;`
For building, a typical tags file is as follows.

```
true : pkg_deriving-ocsigen
<test.ml> : syntax_camlp4o, pkg_deriving-ocsigen.syntax
```

```
type 'a tree =
  | Leaf of 'a
  | Node of 'a * 'a tree * 'a tree
deriving (Show,Eq,Typeable, Functor)

let _ = begin
  print_string (Show.show<int tree> (Node (3, Leaf 4, Leaf 5)));
end
```

5.16 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 comparison, one container*)

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

  for polymorphic set

```
split
union
empty
add
```

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 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

As follows, compare is the right semantics.

```
# 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

Chapter 6

Runtime

1. 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 16
   MByte 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	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 parameters 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 you can have. **this limit arises because of the size of the tag byte and the fact that some of high numbered tags are reserved**
list [1;2;3]	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 special 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).

here we see the module Obj

```
Obj.("gshogh" |> repr |> tag);;
- : int = 252
```

```
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)
```

Chapter 7

GC

1. heap

   Most OCaml blocks are created in the minor(young) heap.

   (a) minor heap (*32K words for 32 bit, 64K for 64 bit by default*) in my mac, i use "ledit ocaml -init x" to avoid loading startup scripts, then

   ```
   Gc.stat ()
   ```

   ```
   {Gc.minor_words = 104194.; Gc.promoted_words = 0.; Gc.major_words = 43979.
    Gc.minor_collections = 0; Gc.major_collections = 0; Gc.heap_words = 12697
    Gc.heap_chunks = 1; Gc.live_words = 43979; Gc.live_blocks = 8446;
    Gc.free_words = 82997; Gc.free_blocks = 1; Gc.largest_free = 82997;
    Gc.fragments = 0; Gc.compactions = 0; Gc.top_heap_words = 126976;
    Gc.stack_size = 52}
   ```

   ```
   78188 lsr 16 ;;
   - : int = 1
   ```

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

```
                             in this direction
```

Consider *the array of two elements*, 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 garbage collector. The fast path just **five machine instructions and no branches**. But even five instructions are costly in inner loops, be careful.

(b) 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, basically copies them by reallocating those objects (recursively) **to the major heap**. After this, any object left in the minor heap **are unreachable**, so the minor heap can be reused by resetting **caml_young_ptr** .

```
           reachable          reachable
+----------+----------+-------+-----+-----+------------------+
|          |//////////|       |///-->///|                    |
+----------+----------+-------+-----+-----+------------------+
           ^                  ^
           |                  |
        local root         local root
```

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.

```
                  "Tag byte space"
+----------+
| 0        | Array, tuple, etc.
+----------+
| 1        |
+----------+
| 2        |
```

```
~          ~
|          | Tags in the range 0..245 are used for variants
~          ~
| 245      |
+----------+
| 246      | Lazy (before being forced)
+----------+
| 247      | Closure
+----------+
| 248      | Object                          ^
+----------+                                 |  Block
   contains
| 249      | Used to implement closures      |  values
   which the
+----------+                                 |  GC should
   scan
| 250      | Used to implement lazy values   |
+----------+ <-------------------------- No_scan_tag
| 251      | Abstract data                   |
+----------+                                 |  Block
   contains
| 252      | String                          |  opaque
   data
+----------+                                 |  which GC
   must
| 253      | Double                          V  not scan
+----------+
| 254      | Array of doubles
+----------+
| 255      | Custom block
+----------+
```

so, in the normal 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 again. 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 sub-passes 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 key-press 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*,

2. 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.

you **should** 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
```

3. 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
```

Chapter 8

Object-oriented

Write later

Chapter 9

complex language features

9.1 stream expression

streams

1. stream expression

```
let rec walk dir =
    let items = try
      Array.map (fun fn -> let path = Filename.concat dir fn in
             try if Sys.is_directory path then `Dir path else `File path
             with e -> `Error(path,e) ) (Sys.readdir dir)
      with e -> [| `Error (dir,e) |] in
      Array.fold_right
        (fun item rest -> match item with
            |`Dir path -> [< 'item ; walk path; rest >]
            | _ -> [< 'item; rest >]) items [< >];;
```

```
val walk :
  string ->
  [> `Dir of string | `Error of string * exn | `File of string ]
  Batteries.Stream.t = <fun>
```

```
(** alternative without syntax extension *)
let rec walk dir =
  let items =
    try
      Array.map
        (fun fn ->
           let path = Filename.concat dir fn   in
             try if Sys.is_directory path
                 then `Dir path
                 else `File path
             with | e -> `Error (path, e))
        (Sys.readdir dir)
    with | e -> [| `Error (dir, e) |]
  in
    Array.fold_right
      (fun item rest ->
         match item with
         | `Dir path ->
             Stream.icons item (Stream.lapp (fun _ -> walk path) rest)
         | _ -> Stream.icons item rest)
      items Stream.sempty

val walk :
  string ->
  [> `Dir of string | `Error of string * exn | `File of string ]
  Batteries.Stream.t = <fun>

Stream.( walk   "/Users/bobzhang1988"
  |> take 10 |> iter
  (
```

```
  (function `Dir s -> "dir :" ^ s
     | `File s -> "file: " ^ s
     | `Error (s,e) -> "error: " ^ s ^ " " ^ Printexc.to_string e
     )  |- print_string |- print_newline)
  );;
```

```
file: /Users/bobzhang1988/#test.el#
file: /Users/bobzhang1988/.bash_history
file: /Users/bobzhang1988/.bashrc
file: /Users/bobzhang1988/.CFUserTextEncoding
file: /Users/bobzhang1988/.DS_Store
file: /Users/bobzhang1988/.emacs
dir :/Users/bobzhang1988/.emacs.d
file: /Users/bobzhang1988/.emacs.d/.emacs
dir :/Users/bobzhang1988/.emacs.d/.git
dir :/Users/bobzhang1988/.emacs.d/.git/branches
```

2. module Stream

```
Stream.npeek;;
- : int -> 'a Batteries.Stream.t -> 'a list = <fun>
Stream.next;;
- : 'a Stream.t -> 'a = <fun>
```

```
let lines_stream_of_channel chan = Stream.from (fun _ ->
    try Some (input_line chan) with End_of_file -> None );;
```

```
val lines_stream_of_channel : BatIO.input -> string Batteries.Stream.t =
```

it raises *Stream.Failure* on an empty stream, i.e. *Stream.next*

```
let line_stream_of_string string =
  Stream.of_list (Str.(split (regexp "\n") string))
```

3. Constructing streams

```
Stream.from
Stream.of_list
Stream.of_string (* char t *)
Stream.of_channel (* char t *)
```

4. Consuming streams

```
Stream.peek
Stream.junk
```

```
let paragraph lines =
  let rec next para_lines i =
    match Stream.peek lines,para_lines with
    | None, [] -> None
    | Some "", [] ->
      Stream.junk lines (* still a white paragraph *)
      next para_lines i
    | Some "", _ | None, _ ->
      Some (String.concat "\n" (List.rev para_lines)) (* a new paragraph*)
    | Some line, _ ->
      Stream.junk lines ;
      next (line :: para_line ) i in
  Stream.from (next [])

let stream_fold f stream init =
    let result = ref init in
    Stream.iter (fun x -> result := f x !result) stre  am; !result;;

val stream_fold : ('a -> 'b -> 'b) -> 'a Batteries.Stream.t -> 'b -> 'b =
  <fun>
```

```
let stream_concat streams =
  let current_stream = ref None in
  let rec next i =
    try
      let stream = match !current_stream with
        | Some stream -> stream
        | None ->
          let stream = Stream.next streams in
          current_stream := Some stream ;
          stream in
      try Some (Stream.next stream)
      with Stream.Failure -> (current_stream := None ; next i)
    with Stream.Failure -> None in
  Stream.from next
```

5. *copying or sharing* streams

   this was called *dup* in Enum

```
(** create 2 buffers to store some pre-fetched value *)
let stream_tee stream =
  let next self other i =
    try
      if Queue.is_empty self
      then
        let value = Stream.next stream in
        Queue.add value other ;
        Some value
      else
        Some (Queue.take self)
    with Stream.Failure -> None in
  let q1,q2 = Queue.create (), Queue.create () in
  (Stream.from (next q1 q2), Stream.from (next q2 q1))
```

6. convert arbitray data types to streams

   if the datat type defines an *iter* function, and you don't mind using threads, you can use a *producer-consumer* arrangement to invert control.

```
let elements iter coll =
  let channel = Event.new_channel () in
  let producer () =
    let _ = iter (fun x -> Event.(sync (send channel (Some x )))) coll in
    Event.(sync (send channel None)) in
  let consumer i =
    Event.(sync (receive channel)) in
  ignore (Thread.create producer ()) ;
  Stream.from consumer

val elements : (('a -> unit) -> 'b -> 'c) -> 'b -> 'a Batteries.Stream.t =
```

   Keep in mind that these techniques spawn producer threads which carry a few risks: they only terminate when they have finished iterating, and any change to the original data structure while iterating may produce unexpected results.

9.2 GADT

```
type _ expr =
  | Int : int -> int expr
  | Add : (int -> int -> int) expr
  | App : ('a -> 'b ) expr * 'a expr -> 'b expr


let rec eval : type t . t expr -> t = function
  | Int n -> n
  | Add -> (+)
  | App (f,x) -> eval f (eval x)
```

```
(** tagless data structure *)
type _ ty =
  | Tint : int ty
  | Tbool : bool ty
  | Tpair : 'a ty * 'b ty -> ('a * 'b) ty

(** inside pattern matching, type inference progresses from left to
    right, allowing subsequent patterns to benift from type equations
    generated in the previous ones.
    This implies that d has type int on the first line,...
*)
let rec print : type a . a ty -> a -> string = fun t d ->
  match t, d with
    |Tint, n -> string_of_int n
    |Tbool,true -> "true"
    |Tbool,false -> "false"
    |Tpair (ta,tb), (a,b) ->
      "(" ^ print ta a ^ ", " ^ print tb b ^ ")"

let f = print (Tpair (Tint,Tbool))
```

9.3 module

read ml 2011 workshop paper

Module can be pased as a value

```
module type ID = sig val id : 'a -> 'a end
let f m =
    let module Id = (val m : ID) in
```

```
   (Id.id 1, Id.id true);;

val f : (module ID) -> int * bool = <fun>

f (module struct let id x = print_endline "ID!"; x end : ID);;

ID!
ID!
```

Here the argument `m` is a module. This is already possible with objects and records, but now modules are also allowed. We introduce three syntaxes

```
(module def : Sig)

(val def : Sig)

(module SIg)
```

```
module type DEVICE = sig end
let devices: (string, (module DEVICE)) Hashtbl.t = Hashtbl.create 18
module PDF = struct end
let _ = Hashtbl.add devices "PDF" (module PDF : DEVICE)
module Device =
  (val (Hashtbl.find devices "PDF") : DEVICE)

module type DEVICE = sig  end
val devices : (string, (module DEVICE)) Batteries.Hashtbl.t = <abstr>
module PDF : sig  end
module Device : DEVICE
```

Runtime choices, Type-safe plugins

Parametric algorithms

Read the slides by Jacques Garrigue

```
module type Number = sig
  type t
  val int : int -> t
  val (+) : t -> t -> t
  val (/) : t -> t -> t
end

let average (type t) number arr =
  let module N = (val number : Number with type t = t) in
  N.(
    let r = ref (int 0) and len = Array.length arr in
    for i = 0 to Pervasives.(len - 1) do
      r := !r + arr.(i)
    done;
    !r / int (Array.length arr)
  )

  val average : (module Number with type t = 'a) -> 'a array -> 'a =
  <fun>

average (module struct type t = int let (+) = (+) let (/) = (/) let int = fun x -
- : int array -> int = <fun>
```

Notice `with type t = int` is necessary here.

9.4 pahantom

jones

jambo

caml

jane

9.5 posit

jane

9.6 private types

Private types

Private type stand between abstract type and concrete types. You can coerce your private type back to the concrete type (zero-performance), but backward is **not allowed**.

For ordinary private type, you can still do pattern match, print the result in toplevel, and debugger. A big advantage for private type abbreviation is that for parameterized type(like container) coercion, you can still do the coercion pretty fast(optimization), and some parameterized types(not containers) can still do such coercions while abstract types can not do. Since ocaml does not provide ad-hoc polymorphism, or type functions like Haskell, this is pretty straight-forward.

```
module Int = struct
  type t = int
  let of_int x = x
  let to_int  x = x
end

module Priv : sig
  type t = private int
  val of_int : int -> t
  val to_int : t -> int
end = Int


module Abstr : sig
  type t
```

```
  val of_int : int -> t
  val to_int : t -> int
end = Int

let _ =
  print_int (Priv.of_int 3 :> int)

let _ =
  List.iter (print_int|-print_newline)
    ([Priv.of_int 1; Priv.of_int 3] :> int list)

(** non-container type *)
type 'a f =
  |A of (int -> 'a)
  |B

(** this is is hard to do when abstract types *)
let  a =
  ((A (fun x -> Priv.of_int x )) :> int f)
```

9.7 Explicit nameing of type variables

The type constructor it introduces can be used in places where a type variable is `not allowed`.

```
let f (type t) () =
    let module M = struct exception E of t end in
    (fun x -> M.E x ), (function M.E x -> Some x | _ -> None);;
val f : unit -> ('a -> exn) * (exn -> 'a option) = <fun>
```

The exception defined in local module can not be captured by other exception handler except wild catch.

Another example:

```
let sort_uniq (type s) (cmp : s -> s -> int) =
    let module S = Set.Make(struct type t = s let compare = cmp end) in
    fun l -> S.elements (List.fold_right S.add l S.empty);;
val sort_uniq : ('a -> 'a -> int) -> 'a list -> 'a list = <fun>
```

The functor needs a type constructor(type variable is not allowed)

9.8 The module Language

Chapter 10

subtle bugs

10.1 Reload duplicate modules

this is fragile when you load some modules like syntax extension, or toploop modules. use *ocamlobjinfo* to see which modules are loaded exactly

Polymorphic comparisons

jane

polymorphic com- pari- son

Chapter 11

interoperating with C

Write later

Chapter 12

Book

12.0.1 Developing Applications with Objective Caml

1. caveat

   (a) + (modulo the boundary, *will not be checked*)

   (b) $1.0/0.0 \to \infty$

   (c) $+. - . * ./. **$ mod ceil floor sqrt exp log log10 cos sin tan acos asin atan

   (d) $asin 3.14 \to nan$

   (e) `char_of_int 255` $\to$ `'\255'` (can not display)

   (f) `char_of_int int_of_char string_of_int int_of_string string_of_int 2551 ->`

   (g) string (length $\leq 2^{24} - 6$)

   (h) == (*physical equal*) (=, != <>)

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

   (i) int * int * int is different from (int * int) * int

   (j) unreasonable parametric equality `(=) : 'a -> 'a -> bool`

(k) 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
   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::pre
```

```
  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
```

(l) 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>
```

(m) 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};;
val add : complex -> complex -> complex = <fun>
```

(n) *redefinition marsks the previous one, while values of the masked types still exist, but it now turns to be an abstract type*

(o) exception

i. `Match_failure Division_by_zero Failure`

ii. exception Name of t – monomorphic , extensible sum Type
when pattern match your exception, its type should be fixed

iii. control flow

(p) **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

2. 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
```

3. 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 *)n
  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

4. library

   (a) 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.
```

   (b) 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
```

(c) 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 =<fu
read_line print_string print_newline print_endline
```

(d) 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
```

(e) stream **imperative**

```
'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 (for my experience, use it in shell, but not in tuareg toplevel)

```
  let concat_stream a b = [<a;b>]
```

```
val concat_stream :
  'a Batteries.Stream.t -> 'a Batteries.Stream.t -> 'a Batteries.Stream.
```

expression not preceded by an c̀onsidered to be sub-stream destructive pattern matching (camlp5 or extended parser can merge) consumed (error), failure

(f) Array List String Hashtbl Buffer Queue

(g) 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
```

(h) Weak (vector of weak pointers) abstract type

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

(i) Printf

```
%t -> (output->unit)
%t%s -> (output->unit)->string->unit
```

they all should be processed at **compile time**

(j) 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
```

(k) 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
```

(l) 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
```

(m) Arg Filename Printexc

(n) 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
```

(o) Num

(p) 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
```

(q) Dynlink

choice at execution time, load a new module and hide the code code (hot-patch) actually (#load is kinda hot-patch), however to write it in programs *more flexible* than #load , load requires 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";;
```

5. syntaxes

6. 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>
```

7. 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
```

8. 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] = m
         | module type module-name = module-type
         | open module-path
         | include module-expr
```

9. 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
```

10. interoperating with C

    Difficutilies

    (a) Machine reperesentation of data
    (b) GC
        calling a c function from ocaml must not modify the memory in ways incompatible with ocaml gc.
    (c) Exceptions
        C does not support exceptions, different mechanisms for aborting computations, this complicates ocaml's exception handling
    (d) sharing common resources
        input-output. each language maintains its own input-output buffers.

    Communications

(a) external declarations

it associates a c function definition with an ocaml name, while giving the type of the latter.

```
external caml_name : type = "C_name"
val caml_name : type
```

both workds, but in the latter case, calls to the c function *first go* through the general function application mechanism of ocaml. This is slightly less efficient, but hides the implementation of the function as a c function.

(b) external functions with more than five arguments

```
external caml_name : type = "C_name_bytecode" "
   C_name_native"
```

chap7 Development Tools

1. Command names

ocaml	toplevel top
ocamlrun	bytecode interpreter
ocamlc	bytecode batch compiler
ocamlopt	native code batch compiler
ocamlc.opt	*optimized* bytecode batch compiler
ocamlopt.opt	*optimized* native code batch compiler
ocamlmktop	new *toplevel* constructor

The optimized compilers are themselves compiled with the Objective Caml native compiler. They compile *faster* but are otherwise *identical* to their unoptimized counterparts.

2. compilation unit

For the interactive system, the unit of compilation corresponds to a phrase of the language. For the batch compiler, the unit of compilation is two files: the source file, and the interface file

extension	meaning
.ml	source
.mli	interface
.cmi	compiled interface
.cmo	object file (byte)
.cma	library object file(bytecode)
.cmx	object file (native)
.cmxa	library object file(native)
.c	c source
.o	c object file (native)
.a	c library object file (native)

The *compiled interface* is used for both the bytecode and native code compiler.

3. ocamlc

-a	construct a runtime library
-c	compile *without* linking
-o name_of_executable	specify the name of the executable
-linkall	link with *all* libraries used
-i	*display all* compiled global declarations
-pp command	preprocessor
-unsafe	turn off index checking
-v	display version
-w list	choose among the list the level of warning message
-impl file	indicate that *file* is a caml source(.ml)
-intf file	as a caml interface(.mli)
-I dir	add directory in the list of directories
-thread	light process
-g, -noassert	linking
-custom, -cclib, -ccopt, -cc	standalone executable
-make-runtime, -use-runtime	runtime
-output-obj	c interface

warning messages.

A/a	enable/disable all messages	the compiler chooses the
F/f	partial application in a sequence	
P/p	incomplete pattern matching	
U/u	missing cases in pattern matching	
X/x	enable/disable all other messages	
M/m and V/v	for hidden object	

(A) by default. turn off some warnings sometimes is helpful, for example

```
ocamlbuild -cflags -w,aPF top_level.cma
```

4. ocamlopt

-compact	optimize the produced code for space
-S	keeps the assembly code in a file
-inline level	set the aggressiveness of inlining

5. Toplevel

-I dir	adds the directory
-unsafe	no bounds checking

6. ocamlmktop

   it's ofen used for pulling native object code libraries(typically written in C) into a new toplevel. *-cclib libname, -ccopt optioin, -custom, -I dir -o exectuable*

```
ocamlmktop -custom -o mytoplevel graphics.cma \
-cclib -I/usr/X11/lib -cclib -lX11
```

This *standalone* exe(-custom) wil be *linked* to the library X11(libX11.a) which in turn will be looked up in the path */usr/X11/lib*

A standalone exe is a program that *does not* depend on OCaml installation to run. The OCaml native compiler produces standalone executables by default. But without *-custom* option, the bytecode compiler produces an executable which requires the *bytecode interpreter ocamlrun*

```
ocamlc test.ml -o a
ocamlc -custom test.ml -o b
```

```
-rwxr-xr-x   1 bob  staff    12225 Dec 23 16:31 a
-rwxr-xr-x   1 bob  staff   198804 Dec 23 16:31 b
```

```
bash-3.2$ cat a | head -n 1
#!/Users/bob/SourceCode/ML/godi/bin/ocamlrun
```

without *-custom*, it depends on *ocamlrun*. With *-custom*, it contains the *Zinc* interpreter as well as the program bytecode, this file can be executed directly or copied to another machien(using the same CPU/Operating System).

Still, the inclusion of machine code means that stand-alone executalbes are not protable to other systems or other architectures.

7. optimization
   It is necessary to not create *intermediate closures* in the case of application on several arguments. For example, when the function *add* is applied with two integers, it is not useful to create the first closure corresponding to the function of applying add to the first argument. It is necessary to note that the creation of a closure would *allocate* certain memory space for the environment and would require the recovery of that memory space in the future. *Automatic memory recovery* is the second major performance concern, along with environment.

8. chap10 Program Analysis Tool

   (a) ocamldep

-I	add dir
-impl,-intf	
-ml(i)-synonym <e>	cosider <e> as a synonym of .ml(i) extension
-modules	Print module dependencies in raw form(not suitable for make)
-native	generate dependencies for a pure native-code project
-slash	for windows & unix

```
ocamldep -modules *.ml
```

```
ta.ml: Array Printf
tb.ml: Array Ta
```

```
ocamldep *.ml
```

```
ta.cmo:
ta.cmx:
tb.cmo: ta.cmo
tb.cmx: ta.cmx
```

other examples

```
ocamlfind ocamldep -modules dir_top_level_util.ml >
   dir_top_level_util.ml.depends
ocamlfind ocamldep -pp 'camlp4of -parser pa_mikmatch_pcre.
   cma' -modules dir_top_level.ml > dir_top_level.ml.
   depends
```

(b) debug

#(un)trace command ,#untrace_all.

```
let verify_div a b q r = a = b * q + r ;;
val verify_div : int -> int -> int -> int -> bool = <fun>
# #trace verif_div ;;
Unbound value verif_div.
# #trace verify_div ;;
verify_div is now traced.
```

```
verify_div 11 5 2 1 ;;

verify_div <-- 11

verify_div --> <fun>

verify_div* <-- 5

verify_div* --> <fun>

verify_div** <-- 2

verify_div** --> <fun>

verify_div*** <-- 1

verify_div*** --> true

- : bool = true
```

```
let rec belongs_to (e:int) = function
    | [] -> false
    | t :: q -> (e=t) || belongs_to e q;;
    val belongs_to : int -> int list -> bool = <fun>
# #trace belongs_to;;
belongs_to is now traced.
# belongs_to 4 [3;5;7;4];;
belongs_to <-- 4
belongs_to --> <fun>
```

```
belongs_to* <-- [3; 5; 7; 4]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [5; 7; 4]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [7; 4]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [4]
belongs_to* --> true
belongs_to* --> true
belongs_to* --> true
belongs_to* --> true
- : bool = true
```

```
# let rec belongs to (e : int) = function
[] -> false
| t :: q -> belongs to e q || (e = t) ; ;
val belongs_to : int -> int list -> bool = <fun> # #trace
   belongs to ;;
belongs_to is now traced.
# belongs to 3 [3;5;7] ;;
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- [3; 5; 7]
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- [5; 7]
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- [7]
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- []
belongs_to* --> false
belongs_to* --> false
belongs_to* --> false
belongs_to* --> true
- : bool = true
```

Trace providing a mechanism for the efficiency analysis of recursive functions, not that friendly, however, no idented output. To make things worse, trace *does not show the value corresponding to an argument of a parameterized type.* The toploop can show only monomorphic types.

Moreover, it only keeps the inferred types of *global declarations.* Therefore after compilation of the expression, the toplevel in fact *no longer* processes any furthuer type information about the expression.
Only global type declarations are kept in the environment of the toplevel loop, *local functions* can not be traced for the same reasons as above

```
let rec belongs_to e = function
    | [] -> false
    | t :: q -> (e=t) || belongs_to e q;;
    val belongs_to : 'a -> 'a list -> bool = <fun>
# belongs_to 4 [3;5;7;4];;
- : bool = true
# #trace belongs_to;;
belongs_to is now traced.
# belongs_to 4 [3;5;7;4];;
belongs_to <-- <poly>
belongs_to --> <fun>
belongs_to* <-- [<poly>; <poly>; <poly>; <poly>]
belongs_to <-- <poly>
belongs_to --> <fun>
belongs_to* <-- [<poly>; <poly>; <poly>]
belongs_to <-- <poly>
belongs_to --> <fun>
belongs_to* <-- [<poly>; <poly>]
belongs_to <-- <poly>
belongs_to --> <fun>
belongs_to* <-- [<poly>]
belongs_to* --> true
belongs_to* --> true
belongs_to* --> true
belongs_to* --> true
- : bool = true
```

(c) ocamldbg

The *-g* option produces a *.cmo* file with the debugging information. (bytecode only)

12.0.2 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

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>
```

```
# 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;;
- : int list = [1; 2; 3; 4]
```

- 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*

12.0.3 caltech ocaml book

(a) 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 **could not** refer to other fields, nor to itself
  - 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, 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 >
exception Dog of dog
exception Cat of cat
let fido : dog = object(self) method v=Dog self method eat = () method
let miao : cat = object(self) method v = Cat self method eat = () meth
```

then 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, this is unlike haskell, always **universal quantifier** required.

```
type 'a animal = <eat:unit; tag : [>] as 'a >;;
(** now we let the compiler to guess the type of 'a *)
let fido : 'a animal = object method eat = () method tag = `Dog 3 end;;
val fido : [> `Dog of int ] animal = <obj>

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

let miao : [> `Cat of int] animal = object method eat = () method tag = `Cat 2 end;;
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

(1) objects (data entirely hidden)

(2) now both are first class (both can be used as arguments)

(3) objects can bind type variable easier, especially 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
class poly = object (self:'self)
method test (x:'self) = x end;;
class poly : object ('a) method test : 'a -> 'a end
# let v = new poly;;
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

– polymorphic class

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

(b) polymorphic variants

(a) simple example

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

```
# 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
```

(b) **define polymorphic variant type**

```
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 ]
```

(c) **sub-typing for polymorphic variants**

```
[`A] :> [`A | `B]
```

since 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>
```

(d) variance notation

if you don't write the + and -, ocaml will **infer** them for you , but when you write abstract type in module type signatures, it makes sense. variance 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

(e) **co-variant** 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>
```

(f) 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>
```

12.0.4 The functional approach to programming

12.0.5 practical ocaml

1. chap30

```
external functions_can_be_defined: unit -> unit = "int_c_code"
```

12.0.6 hol-light

- hol-light

12.1 UNIX system programming in ocaml

12.1.1 chap1

1. Modules Sys and Unix

   Sys containts those functions common to Unix and Windows. **Unix** contains everything specific to Unix.

   The *Sys* and *Unix* modules can override certain functions of the *Pervasives* module

```
Unix.stdin;;
- : Batteries.Unix.file_descr = <abstr>
Pervasives.stdin;;
- : in_channel = <abstr>
```

```
<prog.{native,byte}> : use_unix
ocamlmktop -o ocamlunix unix.cma
```

   When running a program from a shell, the shell passes **arguments** and **environment** to the program. When a program terminates prematurely because *an exception was raised but not caught*, it makes an implicit call to *exit 2*. For *at_exit*, the last function to be registered is called first, and it can not be unregistered. However, we can walk around it using global variables.

```
Sys.argv, Sys.getenv , Unix.environment,
Pervasives.exit, Pervasives.at_exit, Unix.handle_unix_error
```

```
Sys.argv;;
```

```
- : string array =
[|"/Users/bob/SourceCode/ML/godi/bin/ocaml"; "dynlink.cma";
"camlp4of.cma"; "-warn-error"; "+a-4-6-27..29"|]
```

```
Unix.environment ();;
```

```
- : string array =
[|"TERM=dumb"; "SHELL=/bin/bash";
  "TMPDIR=/var/folders/R4/R4awSXDIH6GpuuMmaVeCzU+++TI/-Tmp-/";
  "LIBRARY_PATH=/opt/local/lib/";
  "EMACSDATA=/Applications/Aquamacs.app/Contents/Resources/etc";
  "Apple_PubSub_Socket_Render=/tmp/launch-mcHkKo/Render";
  "EMACSPATH=/Applications/Aquamacs.app/Contents/MacOS/bin";
  "INCLUDE_PATH=/opt/local/include/"; "EMACS=t"; "USER=bob";
  "LD_LIBRARY_PATH=/opt/local/lib/"; "COMMAND_MODE=unix2003"; "TERMCAP=";
  "SSH_AUTH_SOCK=/tmp/launch-g9AcyQ/Listeners";
  "__CF_USER_TEXT_ENCODING=0x1F5:0:0"; "COLUMNS=68";
  "PATH=/opt/local/sbin:/usr/local/smlnj/bin:/usr/local/lib:/Applications/MATI
  "_=/usr/local/bin/ledit"; "C_INCLUDE_PATH=/opt/local/include/";
  "PWD=/Users/bob/SourceCode/Notes/ocaml-book";
  "TEXINPUTS=.:/Applications/Aquamacs.app/Contents/Resources/lisp/aquamacs/edi
  "EMACSLOADPATH=/Applications/Aquamacs.app/Contents/Resources/lisp:/Applicati
  "SHLVL=3"; "HOME=/Users/bob"; "LOGNAME=bob";
  "CAMLP4_EXAMPLE=/Users/bob/SourceCode/ML/godi/build/distfiles/ocaml-3.12.0/c
```

```
"DISPLAY=/tmp/launch-sXEeNT/org.x:0"; "INSIDE_EMACS=23.3.50.1,comint";
"EMACSDOC=/Applications/Aquamacs.app/Contents/Resources/etc";
"SECURITYSESSIONID=616cd3"|]
```

2. ERROR handling

```
exception Unix_error of error * string * string
type error = E2BIG | ... |EUNKNOWERR of int
```

The second arg of *Unix_error* is the name of the system call that raised the error, the third, if possible, identifies the object on which the error occured (i.e. file name). *Unix.handle_unix_error*, if this raises the exception *Unix_error*, displays the message, and *exit 2*

```
let handle_unix_error2 f arg = let open Unix in
  try
     f arg
  with Unix_error(err, fun_name, arg) ->
  prerr_string Sys.argv.(0);
  prerr_string ": \"";
  prerr_string fun_name;
  prerr_string "\" failed";
  if String.length arg > 0 then begin
     prerr_string " on \"";
     prerr_string arg;
     prerr_string "\"" end;
     prerr_string ": ";
     prerr_endline (error_message err);
     exit 2;;
```

```
val handle_unix_error2 : ('a -> 'b) -> 'a -> 'b = <fun>
```

```
let rec restart_on_EINTR f x =
try f x with Unix_error (EINTR, _, _) -> restart_on_EINTR f x
```

```
finally;;
- : (unit -> unit) -> ('a -> 'b) -> 'a -> 'b = <fun>
finally (fun _ -> print_endline "finally") (fun _ -> failwith "haha") ();;
```

```
finally
Exception: Failure "haha".
```

In case the program fails, i.e. raises an exception, *the finalizer is run and the exception ex is raised again.* If **both** the main function and the finalizer fail, the finalizer's exception is raised.

12.1.2 chap2

1. Files

   File covers *standard files, directories, symbolic links, special files(devices), named pipes, sockets*

2. **Filename** module
   makes filename cross platform

```
val current_dir_name : string
val parent_dir_name : string
val dir_sep : string
val concat : string -> string -> string
val is_relative : string -> bool
val is_implicit : string -> bool
val check_suffix : string -> string -> bool
val chop_suffix : string -> string -> string
val chop_extension : string -> string
val basename : string -> string
val dirname : string -> string
val temp_file : ?temp_dir:string -> string -> string ->
   string
val open_temp_file :
  ?mode:open_flag list ->
```

```
    ?temp_dir:string -> string -> string -> string *
      out_channel
  val temp_dir_name : string
  val quote : string -> string
```

non-directory files can have **many parents**(we say that they have many **hard links**). There are also *symbolic links* which can be seen as *non-directory* files containing a path, conceptually, this path can be obtained by reading the contents of the symbolic link like an ordinary file. Whenever a symbolic link occurs in the **middle** of a path, we have to follow its path transparently.

```
  p/s/q -> l/q (l is absolute)
  p/s/q -> p/l/q (l is relative)
```

```
  Sys.getcwd, Sys.chdir, Unix.chroot
```

Unix.chroot p makes the node p, which should be a directory, the root of the *restricted* view of the hierarchy. Absolute paths are then interpreted according to this new root p (and .. at the new root is itself). Due to hard links, a file can have many different names.

```
Unix.(link, unlink,symlink,rename);;

- : (string -> string -> unit) * (string -> unit) *
    (string -> string -> unit) * (string -> string -> unit)
```

unlink f is like *rm -f f*, *link f1 f2* is like *ln f1 f2*, *symlink f1 f2* is like *ln -s f1 f2*, rename f1 f2 is like *mv f1 f2*

A file descriptor represents a pointer to a file along with other information like the current read/write position in the file, the access rights, etc. **file_descr**

```
  Unix.(stdin,stdout,stderr);;

 - : Batteries.Unix.file_descr * Batteries.Unix.file_descr *
  Batteries.Unix.file_descr
```

without redirections, the three descriptors refer to the terminal.

```
cmd > f ; cmd 2 > f
```

3. Meta attributes, types and permissions

```
Unix.(stat,lstat,fstat);;
```

```
(string -> Batteries.Unix.stats) *
(string -> Batteries.Unix.stats) *
(Batteries.Unix.file_descr -> Batteries.Unix.stats)
```

lstat returns information about the symbolic link itself, while *stat* returns information about the file that link points to.

```
Unix.(lstat &&& stat) "/usr/bin/al";;
```

```
({Batteries.Unix.st_dev = 234881026; Batteries.Unix.st_ino = 843893;
  Batteries.Unix.st_kind = Batteries.Unix.S_LNK; (* link *)
  Batteries.Unix.st_perm = 493; Batteries.Unix.st_nlink = 1;
  Batteries.Unix.st_uid = 0; Batteries.Unix.st_gid = 0;
  Batteries.Unix.st_rdev = 0; Batteries.Unix.st_size = 46;
  (* pretty  small as a link *)
  Batteries.Unix.st_atime = 1273804908.;
  Batteries.Unix.st_mtime = 1273804908.;
  Batteries.Unix.st_ctime = 1273804908.},

 {Batteries.Unix.st_dev = 234881026; Batteries.Unix.st_ino = 840746;
  Batteries.Unix.st_kind = Batteries.Unix.S_REG; (*  regular file *)
  Batteries.Unix.st_perm = 493; Batteries.Unix.st_nlink = 1;
  Batteries.Unix.st_uid = 0; Batteries.Unix.st_gid = 80;
```

```
Batteries.Unix.st_rdev = 0; Batteries.Unix.st_size = 163;
(* maybe bigger *)
Batteries.Unix.st_atime = 1323997427.;
Batteries.Unix.st_mtime = 1271968805.;
Batteries.Unix.st_ctime = 1273804911.})
```

A file is uniquely identified by the pair made of its device number(typically the disk partition where it is located) *st_dev* and its inode number *st_ino*
All the users and groups on the machine are usually described in the */etc/-passwd, /etc/groups* files.

```
st_uid
st_gid
getpwnam, getgrnam, (by name, get passwd_entry, group_entry)
getpwuid, getgrgid (by id)
getlogin, getgroups
chown, fchown
```

```
Unix.getlogin () |> Unix.getpwnam;;
```

```
{Batteries.Unix.pw_name = "bob"; Batteries.Unix.pw_passwd = "********";
 Batteries.Unix.pw_uid = 501; Batteries.Unix.pw_gid = 20;
 Batteries.Unix.pw_gecos = "bobzhang"; Batteries.Unix.pw_dir = "/Users/bob";
 Batteries.Unix.pw_shell = "/bin/bash"}
```

for access rights, executable, writable, readable by the user owner, group owner, other users. For a directory, the executable permission means the right to enter it, and read permission the right to list its contents. The special bits do not have meaning unless the **x** bit is set. The bit *t* allows sub-directories to inherit the permissions of the parent directory. On a directory, the bit *s* allows the use of the directory's *uid* or *gid* rather than the user's to create directories. For an executable file, the bit *s* allows the chaning at executation time of the user's effective identity or group with the system calls *setuid* and *setgid*

```
Unix.(setuid, getuid);;
- : (int -> unit) * (unit -> int) = (<fun>, <fun>)
```

4. operations on directries

   only the kernel can write in directories(when files are created). Opening a directory in write mode is *prohibited*.

```
Unix.(opendir,readdir,rewinddir,closedir);;
```

```
- : (string -> Batteries.Unix.dir_handle) *
    (Batteries.Unix.dir_handle -> string) *
    (Batteries.Unix.dir_handle -> unit) * (Batteries.Unix.dir_handle -> unit)
```

   rewinddir repositions the descriptor at the **beginning** of the directory.

```
mkdir, rmdir
```

   We can only remove a directory that is **already empty**. It is thus necessary to first recursively empty the contents of the directory and then remove the directory.

```
exception Hidden of exn
(** add a tag to exn *)
let hide_exn f x = try f x with exn -> raise (Hidden exn)
(** strip the tag of exn *)
let reveal_exn f x = try f x with Hidden exn -> raise exn
```

5. File manipulation

```
Unix.openfile;;
```

```
- : string ->
    Batteries.Unix.open_flag list ->
    Batteries.Unix.file_perm -> Batteries.Unix.file_descr
```

Most programs use *0o666* means *rw-rw-rw-*. with the default creation mask of *0o022*, the file is thus created with the permission *rw-r–r–*. With a more lenient mask of 0o002, the file is created with the permissions *rw-rw-r–*. The third argument can be anything as *O_CREATE* is not specified. And to write to an empty file without caring any previous content, we use

```
Unix.openfile filename [O_WRONLY; O_TRUNC; O_CREAT] 0o666
```

If the file is scripts, we create it with execution permission:

```
Unix.openfile filename [O_WRONLY; O_TRUNC; O_CREAT] 0o777
```

If we want it to be confidential,

```
Unix.openfile filename [O_WRONLY; O_TRUNC; O_CREAT] 0o600
```

The *O_NONBLOCK* flag guarantees that if the file is a named pipe or a special file then the file opening and subsequent reads and writes wil be non-blocking. The *O_NOCTYY* flag guarantees that if the file is a control terminal, it won't become the controlling terminal of the calling process.

```
Unix.(read,single_write);;
```

```
- : (Batteries.Unix.file_descr -> string -> int -> int -> int) *
    (Batteries.Unix.file_descr -> string -> int -> int -> int)
```

The *string* hold the read bytes or the bytes to write. The 3rd argument is the start, the forth is the number.

For writes, the number of bytes actually written is usually the number of bytes requested, with two exceptions (i) not possible to write (i.e. disk is full) (ii) the descript is a pipe or a socket open in non-blocking mode(async) (iii) due to OCaml, too large.

The reason for (iii) is that internally OCaml uses auxiliary buffer whose size is bounded by a maximal value.

OCaml also provides *Unix.write* which iterates the writes until all the data is written or an error occurs. The problem is that in case of error there's no way to know the number of bytes that were *actually written*. *single_write* preserves the atomicity of writes.

For reads, when the current position is at the end of file, read returns zero. The convention *zero equals end of file* also holds for special files, *i.e. pipes and sockets*. For example, read on a terminal returns zero if we issue a *Ctrl-D* on the input.

But you may consider the blocking-mode in case.

```
Unix.close : file_descr -> unit
```

In contrast to Pervasives' channels, a file descriptor does not need to be closed to ensure that all pending writes have been performed as write requests are *immediately* transmitted to the kernel. On the other hand, the number of descriptors allocated by a process is limited by the kernel(several hundreds to thousands).

```
let buffer_size = 8192
let buffer = String.create buffer_size

(** this is unsatisfactory, if we copy an executable file, we would
like the copy to be also executable. *)
let file_copy input output = Unix.(
  let fd_in = openfile input [O_RDONLY] 0 in
  let fd_out = openfile output [O_WRONLY; O_CREAT; O_TRUNC] 0o666 in
  let rec copy_loop () = match read fd_in buffer 0 buffer_size with
```

```
    |0 -> ()
    |r -> write fd_out buffer 0 r |> ignore; copy_loop () in
  copy_loop ();
  close fd_in ;
  close fd_out
)


let copy () =
  if Array.length Sys.argv = 3 then begin
    file_copy Sys.argv.(1) Sys.argv.(2)
  end
  else begin
    prerr_endline
      ("Usage: " ^ Sys.argv.(0) ^ "<input_file> <output_file>");
    exit 1
  end

let _  = Unix.handle_unix_error copy ()
```

```
ocamlbuild find.byte -- find.ml find.xxxx
```

```
ocamlbuild find.byte -- find.mlx find.xxxx
_build/find.byte: "open" failed on "find.mlx": No such file or directory
```

6. system call

   For a system call, even if it does very little work, cost dearly – much more than a normal function call. So we need buffer to reduce the number of system call. For ocaml, the *Pervasives* module adds another layer *in_channel, out_channel*.

7. positioning and operations specific to certain file types

```
Unix.lseek;;
- : Batteries.Unix.file_descr -> int -> Batteries.Unix.seek_command -> int =
```

File descriptors provide a uniform and media-independent interface for data communicatioin. However this uniformity breaks when we need to access all the features provided by a given media.

For normal files, specific API

```
Unix.(truncate,ftruncate);;
- : (string -> int -> unit) * (Batteries.Unix.file_descr -> int -> unit) =
```

For symbolic links

```
Unix.(symlink, readlink);;
- : (string -> string -> unit) * (string -> string) = (<fun>, <fun>)
```

special files

(a) /dev/null black hole. (useful for ignoring the result)

(b) /dev/tty* control terminals

(c) /dev/pty* pseudo-terminals

(d) /dev/hd* disks

(e) /proc Under linux, system parameters organized as a file system.

many special files ignore *lseek*

8. terminals

```
Unix.(tcgetattr, tcsetattr);;
```

```
(Batteries.Unix.file_descr -> Batteries.Unix.terminal_io) *
(Batteries.Unix.file_descr ->
    Batteries.Unix.setattr_when -> Batteries.Unix.terminal_io -> unit)
```

```
Unix.(tcgetattr stdout);;
```

```
{Batteries.Unix.c_ignbrk = false; Batteries.Unix.c_brkint = true;
 Batteries.Unix.c_ignpar = false; Batteries.Unix.c_parmrk = false;
 Batteries.Unix.c_inpck = false; Batteries.Unix.c_istrip = false;
 Batteries.Unix.c_inlcr = false; Batteries.Unix.c_igncr = false;
 Batteries.Unix.c_icrnl = true; Batteries.Unix.c_ixon = false;
 Batteries.Unix.c_ixoff = false; Batteries.Unix.c_opost = true;
 Batteries.Unix.c_obaud = 9600; Batteries.Unix.c_ibaud = 9600;
 Batteries.Unix.c_csize = 8; Batteries.Unix.c_cstopb = 1;
 Batteries.Unix.c_cread = true; Batteries.Unix.c_parenb = false;
 Batteries.Unix.c_parodd = false; Batteries.Unix.c_hupcl = true;
 Batteries.Unix.c_clocal = false; Batteries.Unix.c_isig = false;
 Batteries.Unix.c_icanon = false; Batteries.Unix.c_noflsh = false;
 Batteries.Unix.c_echo = false; Batteries.Unix.c_echoe = true;
 Batteries.Unix.c_echok = false; Batteries.Unix.c_echonl = false;
 Batteries.Unix.c_vintr = '\003'; Batteries.Unix.c_vquit = '\028';
 Batteries.Unix.c_verase = '\255'; Batteries.Unix.c_vkill = '\255';
 Batteries.Unix.c_veof = '\004'; Batteries.Unix.c_veol = '\255';
 Batteries.Unix.c_vmin = 1; Batteries.Unix.c_vtime = 0;
 Batteries.Unix.c_vstart = '\017'; Batteries.Unix.c_vstop = '\019'}
```

it seems that ledit will change your input, and you can not get *Unix.(tcgetattr stdin)* work.

The code below works in real terminal, but does not work in pseudo-terminals(like Emacs)

```
let read_passwd message = Unix.(
match
   try
    let default = tcgetattr stdin in
```

```
    let silent = {default with c_echo = false; c_echoe = false ;
                  c_echok = false; c_echonl = false ; } in
     Some (default, silent)
   with _ -> None
with
 |None -> Legacy.input_line Pervasives.stdin
 |Some (default, silent) ->
   print_string message ;
   Legacy.flush Pervasives.stdout ;
   tcsetattr stdin TCSANOW silent ;
   try
     let s = Legacy.input_line Pervasives.stdin in
     tcsetattr stdin TCSANOW default; s
   with x ->      tcsetattr stdin TCSANOW default; raise x

);;
```

Sometimes a program needs to start another and connect its standard input to a terminal (or pseudo-terminal). To achieve that, we must manually look among the pseudo-terminals(/dev/tty[a-z][a-f0-9]) and find one that is not already open. We can open this file and start the program with this file on its standard input.

The function *tcsendbreak* sends an interrupt to the peripheral. The second argument is the duration of the interrupt.

```
   tcdrain, tcflush, tcflow, setsid
```

9. locks on files

```
Unix.lockf;;
- : Batteries.Unix.file_descr -> Batteries.Unix.lock_command ->
   int -> unit =
```

ocaml-expect

```
let p = X.spawn "ocaml" [||];;
val p : X.t = <abstr>
X.expect p ~fmatches:[(fun s -> Some s)] [] "";;
- : string = "        Objective Caml version 3.12.1"
X.send p "3;;\n";;
- : unit = ()
X.expect p ~fmatches:[(fun s -> Some s)] [] "";;
- : string = "- : int = 3"
```

not very powerful

12.1.3 chap3

12.1.4 practical ocaml

1. chap30

```
external functions_can_be_defined: unit -> unit = "int_c_code"
```

12.1.5 tricks

- ocamlobjinfo

  analyzing 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:

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

obj has many Units, each Unit itself also import some interfaces. ideas: you can parse the result to get an dependent graph.

- 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 intentionally omitted

  this is a new feature in recent ocaml, it will emit an warning otherwise

- 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

- **dirty** 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)
;;
```

12.1.6 ocaml blogs

ygrek

michal

eigenclass

syntax

jambon

Xavier Clerc

Zheng li

xleroy/teaching

alaska

erratique

duther

David Teller

john harisson

Mike Gordon

Robert Keller

alexott

Yoann Padioleau

garrigue

jun

llvm

incubaid

heniz