Preface

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

☺

Acknowledgements

write later

Contents

Todo list

Chapter 1

Tool Chain

Chapter 2

Lexing

Chapter 3

Parsing

Chapter 4

Camlp4

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

4.3 Camlp4 SourceCode Exploration

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
|       `-- Grammar
|-- Camlp4Filters
|-- Camlp4Parsers
|-- Camlp4Printers
|-- Camlp4Top
|-- examples
|-- man
|-- test
|   `-- fixtures
`-- unmaintained
```

4.3.1 Camlp4.Sig

For `Camlp4.Sig.ml`, all are signatures. It's convention for ocaml programmers to document most signatures in *Sig.ml*.

Basic Signature

```
module type Type = sig
  type t;
end;
```

Listing 26: Signature with a type

```
module type Error = sig
  type t;
  exception E of t;
  value to_string : t -> string;
  value print : Format.formatter -> t -> unit;
end;
```

Listing 27: Error module type

Signature for errors modules, an *Error* module can be registred with the *ErrorHandler.Register* functor in order to be well printed.

```
module Warning (Loc : Type) = struct
  module type S = sig
    type warning = Loc.t -> string -> unit;
    value default_warning : warning;
    value current_warning : ref warning;
    value print_warning   : warning;
  end;
end;
```

Listing 28: Warning Functor

It's interesting here *Warning* is a type level function actually. Making it a function so you can use *Warning(Loc).S* later.

Advanced Signatures

Loc A signature for locations

Ast Ast *minimal, abstract* signature

Camlp4Ast Ast *concrete* signature

Camlp4AstToAst *functor (M:Camp4Ast) : Ast with type .. = ...*. This functor is a restriction functor. You can use it like this *with module Ast = Camlp4.Sig.Camlp4AstToAst Camlp4Ast*

MakeCamlp4Ast the only concrete definition of camlp4 ast. You can write some generic plugins here.

AstFilters Registering and folding of Ast filters, it includes *Camlp4Ast* and some filter functions

DynAst Asts as one single dynamic type

Quotation signature for a quoation expander registery

Token A signature around tokens.

Camlp4Token *Token with type t = camlp4_token*

DynLoader

Grammar

Lexer

Parser Parser is a type lever function, like *Warning*

Printer The same as above

Syntax A syntax module is a sort of constistent bunch of modules and values. In such a module you have a parser, a printer, and also modules for locations, syntax trees, tokens, grammars, quotations, anti-quotations. There is also the main grammar entries.

Camlp4Syntax Ast is replaced with Camlp4Ast

SyntaxExtension functor signature.

```
module type SyntaxExtension = functor (Syn : Syntax)
                    -> (Syntax with module Loc            = Syn.Loc
                                and module Ast            = Syn.Ast
                                and module Token          = Syn.Token
                                and module Gram           = Syn.Gram
                                and module Quotation      = Syn.Quotation);
```

We can make use of these signatures to write signature file. *Ast* is an

4.3.2 Camlp4.PreCast

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

File `Camlp4.PreCast` Listing ?? packed such modules

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

```
(** 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 (* *, +, +++*%, %#@... *)
  | LIDENT        of string (* lower case identifier *)
  | UIDENT        of string (* upper case identifier *)
  | ESCAPED_IDENT of string (* ( * ), ( ++##> ), ( foo ) *)
  | INT           of int and string  (* `INT(i,is)  42, 0xa0, 0XffFFff, 0b1010101, 0O644, 0o644 *)
  | INT32         of int32 and string (* 42l, 0xa0l... *)
  | INT64         of int64 and string (* 42L, 0xa0L... *)
  | NATIVEINT     of nativeint and string (* 42n, 0xa0n... *)
  | FLOAT         of float and string (* 42.5, 1.0, 2.4e32 *)
  | CHAR          of char and string  (* with escaping *)
  | STRING        of string and string (* with escaping *)
  | LABEL         of string (* *)
  | OPTLABEL      of string
  | QUOTATION     of Sig.quotation (* << foo >> <:quot_name< bar >> <@loc_name<bar>>
     type quotation ={ q_name : string; q_loc : string;
     q_shift : int; q_contents : string; } *)
  | ANTIQUOT      of string and string    (* $foo$  $anti_name:foo$ $`anti_name:foo$ *)
  | COMMENT       of string
  | BLANKS        of string               (* some non newline blanks *)
  | NEWLINE                               (* interesting *)
  | LINE_DIRECTIVE of int and option string  (* #line 42, #foo "string" *)
  | 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. We will give a
    nice solution later *)
```

```
module MakeSyntax (U : sig end) = OCamlInitSyntax.Make Ast Gram Quotation;
module Syntax = MakeSyntax (struct end);
module AstFilters = Struct.AstFilters.Make Ast; (** Functorize *)
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;
```

Listing 29: Camlp4 PreCast

4.3.3 Camlp4.OCamlInitSyntax

`OCamlInitSyntax` Listing 30 does not do too many things, first, it initialize all the entries needed later (they are all blank, to be extended by your functor), after initialization, it created a submodule `AntiquotSyntax`, and initialize two entries `antiquot_expr` and `antiquot_patt`, very easy.

Its signature is as follows:

```
module 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
                   and module Ast = Ast
                   and module Token = Gram.Token
                   and module Gram = Gram
                   and module Quotation = Quotation
```

```
(** File Camlp4.OCamlInitSyntax.ml
    Ast -> Gram -> Quotation -> Camlp4Syntax
    Given Ast, Gram, Quotation, we produce Camlp4Syntax
 *)
module 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
                   and module Ast = Ast
                   and module Token = Gram.Token
                   and module Gram = Gram
                   and module Quotation = Quotation
= struct

  module Loc     = Ast.Loc;
  module Ast     = Ast;
  module Gram    = Gram;
  module Token   = Gram.Token;
  open Sig;

  (* Warnings *)
  type warning = Loc.t -> string -> unit;
  value default_warning loc txt = Format.eprintf "<W> %a: %s@." Loc.print loc txt;
  value current_warning = ref default_warning;
  value print_warning loc txt = current_warning.val loc txt;

  value a_CHAR = Gram.Entry.mk "a_CHAR";
  value a_FLOAT = Gram.Entry.mk "a_FLOAT";
  value a_INT = Gram.Entry.mk "a_INT";
  value a_INT32 = Gram.Entry.mk "a_INT32";
  value a_INT64 = Gram.Entry.mk "a_INT64";
  value a_LABEL = Gram.Entry.mk "a_LABEL";
  value a_LIDENT = Gram.Entry.mk "a_LIDENT";
  value a_NATIVEINT = Gram.Entry.mk "a_NATIVEINT";
  value a_OPTLABEL = Gram.Entry.mk "a_OPTLABEL";
  value a_STRING = Gram.Entry.mk "a_STRING";
  value a_UIDENT = Gram.Entry.mk "a_UIDENT";
  value a_ident = Gram.Entry.mk "a_ident";
  value amp_ctyp = Gram.Entry.mk "amp_ctyp";
  value and_ctyp = Gram.Entry.mk "and_ctyp";
  value match_case = Gram.Entry.mk "match_case";
  value match_case0 = Gram.Entry.mk "match_case0";
  value binding = Gram.Entry.mk "binding";
  value class_declaration = Gram.Entry.mk "class_declaration";
  value class_description = Gram.Entry.mk "class_description";
  value class_expr = Gram.Entry.mk "class_expr";
  value class_fun_binding = Gram.Entry.mk "class_fun_binding";
  value class_fun_def = Gram.Entry.mk "class_fun_def";
  value class_info_for_class_expr = Gram.Entry.mk "class_info_for_class_expr";
  value class_info_for_class_type = Gram.Entry.mk "class_info_for_class_type";
  value class_longident = Gram.Entry.mk "class_longident";
  value class_longident_and_param = Gram.Entry.mk "class_longident_and_param";
```

```
value class_name_and_param = Gram.Entry.mk "class_name_and_param";
value class_sig_item = Gram.Entry.mk "class_sig_item";
value class_signature = Gram.Entry.mk "class_signature";
value class_str_item = Gram.Entry.mk "class_str_item";
value class_structure = Gram.Entry.mk "class_structure";
value class_type = Gram.Entry.mk "class_type";
value class_type_declaration = Gram.Entry.mk "class_type_declaration";
value class_type_longident = Gram.Entry.mk "class_type_longident";
value class_type_longident_and_param = Gram.Entry.mk "class_type_longident_and_param";
value class_type_plus = Gram.Entry.mk "class_type_plus";
value comma_ctyp = Gram.Entry.mk "comma_ctyp";
value comma_expr = Gram.Entry.mk "comma_expr";
value comma_ipatt = Gram.Entry.mk "comma_ipatt";
value comma_patt = Gram.Entry.mk "comma_patt";
value comma_type_parameter = Gram.Entry.mk "comma_type_parameter";
value constrain = Gram.Entry.mk "constrain";
value constructor_arg_list = Gram.Entry.mk "constructor_arg_list";
value constructor_declaration = Gram.Entry.mk "constructor_declaration";
value constructor_declarations = Gram.Entry.mk "constructor_declarations";
value ctyp = Gram.Entry.mk "ctyp";
value cvalue_binding = Gram.Entry.mk "cvalue_binding";
value direction_flag = Gram.Entry.mk "direction_flag";
value direction_flag_quot = Gram.Entry.mk "direction_flag_quot";
value dummy = Gram.Entry.mk "dummy";
value entry_eoi = Gram.Entry.mk "entry_eoi";
value eq_expr = Gram.Entry.mk "eq_expr";
value expr = Gram.Entry.mk "expr";
value expr_eoi = Gram.Entry.mk "expr_eoi";
value field_expr = Gram.Entry.mk "field_expr";
value field_expr_list = Gram.Entry.mk "field_expr_list";
value fun_binding = Gram.Entry.mk "fun_binding";
value fun_def = Gram.Entry.mk "fun_def";
value ident = Gram.Entry.mk "ident";
value implem = Gram.Entry.mk "implem";
value interf = Gram.Entry.mk "interf";
value ipatt = Gram.Entry.mk "ipatt";
value ipatt_tcon = Gram.Entry.mk "ipatt_tcon";
value label = Gram.Entry.mk "label";
value label_declaration = Gram.Entry.mk "label_declaration";
value label_declaration_list = Gram.Entry.mk "label_declaration_list";
value label_expr = Gram.Entry.mk "label_expr";
value label_expr_list = Gram.Entry.mk "label_expr_list";
value label_ipatt = Gram.Entry.mk "label_ipatt";
value label_ipatt_list = Gram.Entry.mk "label_ipatt_list";
value label_longident = Gram.Entry.mk "label_longident";
value label_patt = Gram.Entry.mk "label_patt";
value label_patt_list = Gram.Entry.mk "label_patt_list";
```

```
value labeled_ipatt = Gram.Entry.mk "labeled_ipatt";
value let_binding = Gram.Entry.mk "let_binding";
value meth_list = Gram.Entry.mk "meth_list";
value meth_decl = Gram.Entry.mk "meth_decl";
value module_binding = Gram.Entry.mk "module_binding";
value module_binding0 = Gram.Entry.mk "module_binding0";
value module_declaration = Gram.Entry.mk "module_declaration";
value module_expr = Gram.Entry.mk "module_expr";
value module_longident = Gram.Entry.mk "module_longident";
value module_longident_with_app = Gram.Entry.mk "module_longident_with_app";
value module_rec_declaration = Gram.Entry.mk "module_rec_declaration";
value module_type = Gram.Entry.mk "module_type";
value package_type = Gram.Entry.mk "package_type";
value more_ctyp = Gram.Entry.mk "more_ctyp";
value name_tags = Gram.Entry.mk "name_tags";
value opt_as_lident = Gram.Entry.mk "opt_as_lident";
value opt_class_self_patt = Gram.Entry.mk "opt_class_self_patt";
value opt_class_self_type = Gram.Entry.mk "opt_class_self_type";
value opt_class_signature = Gram.Entry.mk "opt_class_signature";
value opt_class_structure = Gram.Entry.mk "opt_class_structure";
value opt_comma_ctyp = Gram.Entry.mk "opt_comma_ctyp";
value opt_dot_dot = Gram.Entry.mk "opt_dot_dot";
value row_var_flag_quot = Gram.Entry.mk "row_var_flag_quot";
value opt_eq_ctyp = Gram.Entry.mk "opt_eq_ctyp";
value opt_expr = Gram.Entry.mk "opt_expr";
value opt_meth_list = Gram.Entry.mk "opt_meth_list";
value opt_mutable = Gram.Entry.mk "opt_mutable";
value mutable_flag_quot = Gram.Entry.mk "mutable_flag_quot";
value opt_polyt = Gram.Entry.mk "opt_polyt";
value opt_private = Gram.Entry.mk "opt_private";
value private_flag_quot = Gram.Entry.mk "private_flag_quot";
value opt_rec = Gram.Entry.mk "opt_rec";
value rec_flag_quot = Gram.Entry.mk "rec_flag_quot";
value opt_sig_items = Gram.Entry.mk "opt_sig_items";
value opt_str_items = Gram.Entry.mk "opt_str_items";
value opt_virtual = Gram.Entry.mk "opt_virtual";
value virtual_flag_quot = Gram.Entry.mk "virtual_flag_quot";
value opt_override = Gram.Entry.mk "opt_override";
value override_flag_quot = Gram.Entry.mk "override_flag_quot";
value opt_when_expr = Gram.Entry.mk "opt_when_expr";
value patt = Gram.Entry.mk "patt";
value patt_as_patt_opt = Gram.Entry.mk "patt_as_patt_opt";
value patt_eoi = Gram.Entry.mk "patt_eoi";
value patt_tcon = Gram.Entry.mk "patt_tcon";
value phrase = Gram.Entry.mk "phrase";
value poly_type = Gram.Entry.mk "poly_type";
value row_field = Gram.Entry.mk "row_field";
```

```
value sem_expr = Gram.Entry.mk "sem_expr";
value sem_expr_for_list = Gram.Entry.mk "sem_expr_for_list";
value sem_patt = Gram.Entry.mk "sem_patt";
value sem_patt_for_list = Gram.Entry.mk "sem_patt_for_list";
value semi = Gram.Entry.mk "semi";
value sequence = Gram.Entry.mk "sequence";
value do_sequence = Gram.Entry.mk "do_sequence";
value sig_item = Gram.Entry.mk "sig_item";
value sig_items = Gram.Entry.mk "sig_items";
value star_ctyp = Gram.Entry.mk "star_ctyp";
value str_item = Gram.Entry.mk "str_item";
value str_items = Gram.Entry.mk "str_items";
value top_phrase = Gram.Entry.mk "top_phrase";
value type_constraint = Gram.Entry.mk "type_constraint";
value type_declaration = Gram.Entry.mk "type_declaration";
value type_ident_and_parameters = Gram.Entry.mk "type_ident_and_parameters";
value type_kind = Gram.Entry.mk "type_kind";
value type_longident = Gram.Entry.mk "type_longident";
value type_longident_and_parameters = Gram.Entry.mk "type_longident_and_parameters";
value type_parameter = Gram.Entry.mk "type_parameter";
value type_parameters = Gram.Entry.mk "type_parameters";
value typevars = Gram.Entry.mk "typevars";
value use_file = Gram.Entry.mk "use_file";
value val_longident = Gram.Entry.mk "val_longident";
value value_let = Gram.Entry.mk "value_let";
value value_val = Gram.Entry.mk "value_val";
value with_constr = Gram.Entry.mk "with_constr";
value expr_quot = Gram.Entry.mk "quotation of expression";
value patt_quot = Gram.Entry.mk "quotation of pattern";
value ctyp_quot = Gram.Entry.mk "quotation of type";
value str_item_quot = Gram.Entry.mk "quotation of structure item";
value sig_item_quot = Gram.Entry.mk "quotation of signature item";
value class_str_item_quot = Gram.Entry.mk "quotation of class structure item";
value class_sig_item_quot = Gram.Entry.mk "quotation of class signature item";
value module_expr_quot = Gram.Entry.mk "quotation of module expression";
value module_type_quot = Gram.Entry.mk "quotation of module type";
value class_type_quot = Gram.Entry.mk "quotation of class type";
value class_expr_quot = Gram.Entry.mk "quotation of class expression";
value with_constr_quot = Gram.Entry.mk "quotation of with constraint";
value binding_quot = Gram.Entry.mk "quotation of binding";
value rec_binding_quot = Gram.Entry.mk "quotation of record binding";
value match_case_quot = Gram.Entry.mk "quotation of match_case (try/match/function case)";
value module_binding_quot = Gram.Entry.mk "quotation of module rec binding";
value ident_quot = Gram.Entry.mk "quotation of identifier";
value prefixop = Gram.Entry.mk "prefix operator (start with '!', '?', '~')";
value infixop0 = Gram.Entry.mk "infix operator (level 0) (comparison operators, and some others)";
value infixop1 = Gram.Entry.mk "infix operator (level 1) (start with '^', '@')";
```

```
value infixop2 = Gram.Entry.mk "infix operator (level 2) (start with '+', '-')";
value infixop3 = Gram.Entry.mk "infix operator (level 3) (start with '*', '/', '%')";
value infixop4 = Gram.Entry.mk "infix operator (level 4) (start with \"**\") (right assoc)";

EXTEND Gram
  top_phrase:
    [ [ `EOI -> None ] ]
  ;
END;

module AntiquotSyntax = struct
  module Loc  = Ast.Loc;
  module Ast  = Sig.Camlp4AstToAst Ast;
  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 wrap directive_handler pa init_loc cs =
  let rec loop loc =
    let (pl, stopped_at_directive) = pa loc cs in
    match stopped_at_directive with
    [ Some new_loc ->
      let pl =
        match List.rev pl with
        [ [] -> assert False
        | [x :: xs] ->
            match directive_handler x with
            [ None -> xs
            | Some x -> [x :: xs] ] ]
      in (List.rev pl) @ (loop new_loc)
    | None -> pl ]
  in loop init_loc;

value parse_implem ?(directive_handler = fun _ -> None) _loc cs =
```

```
    let l = wrap directive_handler (Gram.parse implem) _loc cs in
    <:str_item< $list:l$ >>;

  value parse_interf ?(directive_handler = fun _ -> None) _loc cs =
    let l = wrap directive_handler (Gram.parse interf) _loc cs in
    <:sig_item< $list:l$ >>;

  value print_interf ?input_file:(_) ?output_file:(_) _ = failwith "No interface printer";
  value print_implem ?input_file:(_) ?output_file:(_) _ = failwith "No implementation printer";
end;
```

Listing 30: OCamlInitSyntax

4.3.4 Camlp4.Struct.Camlp4Ast.mlast

This file use macro`INCLUDE` to include `Camlp4.Camlp4Ast.parital.ml` for reuse.

4.3.5 Camlp4.Register.AstFilter

Notice an interesting module `AstFilter` Listing 31, is defined by `Struct.AstFilters.Make`, which we see in `Camlp4.PreCast.ml`?? It's very simple actually.

```
module AstFilter
  (Id : Sig.Id) (Maker : functor (F : Sig.AstFilters) -> sig end) =
struct
  declare_dyn_module Id.name (fun _ -> let module M = Maker AstFilters in ());
```

```
(**AstFilters.ml*)
module Make (Ast : Sig.Camlp4Ast)
: Sig.AstFilters with module Ast = Ast
= struct

  module Ast = Ast;

  type filter 'a = 'a -> 'a;

  value interf_filters = Queue.create ();
  value fold_interf_filters f i = Queue.fold f i interf_filters;
  value implem_filters = Queue.create ();
  value fold_implem_filters f i = Queue.fold f i implem_filters;
  value topphrase_filters = Queue.create ();
  value fold_topphrase_filters f i = Queue.fold f i topphrase_filters;
```

```
  value register_sig_item_filter f = Queue.add f interf_filters;
  value register_str_item_filter f = Queue.add f implem_filters;
  value register_topphrase_filter f = Queue.add f topphrase_filters;
end;
```

Listing 31: AstFilters

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

Listing 32: Camlp4Ast Make

4.3.6 Camlp4.Register

Let's see what's in `Register` Listing 33module

```
(**
   Camlp4.Register.ml
*)
module PP = Printers;
open PreCast;

type parser_fun 'a =
  ?directive_handler:('a -> option 'a) -> PreCast.Loc.t -> Stream.t char -> 'a;

type printer_fun 'a =
  ?input_file:string -> ?output_file:string -> 'a -> unit;

(** a lot of parsers to be modified *)
value sig_item_parser = ref (fun ?directive_handler:(_) _ _ -> failwith "No interface parser");
value str_item_parser = ref (fun ?directive_handler:(_) _ _ -> failwith "No implementation parser");

value sig_item_printer = ref (fun ?input_file:(_) ?output_file:(_) _ -> failwith "No interface printer");
value str_item_printer = ref (fun ?input_file:(_) ?output_file:(_) _ -> failwith "No implementation printer");

(** a queue of callbacks *)
value callbacks = Queue.create ();

value loaded_modules = ref [];
```

```
(** iterate each callback*)
value iter_and_take_callbacks f =
  let rec loop () = loop (f (Queue.take callbacks)) in
  try loop () with [ Queue.Empty -> () ];

(** register module, add to the Queue *)
value declare_dyn_module (m:string) (f:unit->unit) =
  begin
    (* let () = Format.eprintf "declare_dyn_module: %s@." m in *)
    loaded_modules.val := [ m :: loaded_modules.val ];
    Queue.add (m, f) callbacks;
  end;

value register_str_item_parser f = str_item_parser.val := f;
value register_sig_item_parser f = sig_item_parser.val := f;
value register_parser f g =
  do { str_item_parser.val := f; sig_item_parser.val := g };
value current_parser () = (str_item_parser.val, sig_item_parser.val);

value register_str_item_printer f = str_item_printer.val := f;
value register_sig_item_printer f = sig_item_printer.val := f;
value register_printer f g =
  do { str_item_printer.val := f; sig_item_printer.val := g };
value current_printer () = (str_item_printer.val, sig_item_printer.val);

module Plugin (Id : Sig.Id) (Maker : functor (Unit : sig end) -> sig end) = struct
  declare_dyn_module Id.name (fun _ -> let module M = Maker (struct end) in ());
end;

module SyntaxExtension (Id : Sig.Id) (Maker : Sig.SyntaxExtension) = struct
  declare_dyn_module Id.name (fun _ -> let module M = Maker Syntax in ());
end;

module OCamlSyntaxExtension
  (Id : Sig.Id) (Maker : functor (Syn : Sig.Camlp4Syntax) -> Sig.Camlp4Syntax) =
struct
  declare_dyn_module Id.name (fun _ -> let module M = Maker Syntax in ());
end;

module SyntaxPlugin (Id : Sig.Id) (Maker : functor (Syn : Sig.Syntax) -> sig end) = struct
  declare_dyn_module Id.name (fun _ -> let module M = Maker Syntax in ());
end;

module Printer
  (Id : Sig.Id) (Maker : functor (Syn : Sig.Syntax)
                                -> (Sig.Printer Syn.Ast).S) =
struct
```

```
  declare_dyn_module Id.name (fun _ ->
    let module M = Maker Syntax in
    register_printer M.print_implem M.print_interf);
end;

module OCamlPrinter
  (Id : Sig.Id) (Maker : functor (Syn : Sig.Camlp4Syntax)
                                -> (Sig.Printer Syn.Ast).S) =
struct
  declare_dyn_module Id.name (fun _ ->
    let module M = Maker Syntax in
    register_printer M.print_implem M.print_interf);
end;

module OCamlPreCastPrinter
  (Id : Sig.Id) (P : (Sig.Printer PreCast.Ast).S) =
struct
  declare_dyn_module Id.name (fun _ ->
    register_printer P.print_implem P.print_interf);
end;

module Parser
  (Id : Sig.Id) (Maker : functor (Ast : Sig.Ast)
                                -> (Sig.Parser Ast).S) =
struct
  declare_dyn_module Id.name (fun _ ->
    let module M = Maker PreCast.Ast in
    register_parser M.parse_implem M.parse_interf);
end;

module OCamlParser
  (Id : Sig.Id) (Maker : functor (Ast : Sig.Camlp4Ast)
                                -> (Sig.Parser Ast).S) =
struct
  declare_dyn_module Id.name (fun _ ->
    let module M = Maker PreCast.Ast in
    register_parser M.parse_implem M.parse_interf);
end;

module OCamlPreCastParser
  (Id : Sig.Id) (P : (Sig.Parser PreCast.Ast).S) =
struct
  declare_dyn_module Id.name (fun _ ->
    register_parser P.parse_implem P.parse_interf);
end;

module AstFilter
```

```
  (Id : Sig.Id) (Maker : functor (F : Sig.AstFilters) -> sig end) =
struct
  declare_dyn_module Id.name (fun _ -> let module M = Maker AstFilters in ());
end;

sig_item_parser.val := Syntax.parse_interf;
str_item_parser.val := Syntax.parse_implem;

module CurrentParser = struct
  module Ast = Ast;
  value parse_interf ?directive_handler loc strm =
    sig_item_parser.val ?directive_handler loc strm;
  value parse_implem ?directive_handler loc strm =
    str_item_parser.val ?directive_handler loc strm;
end;

module CurrentPrinter = struct
  module Ast = Ast;
  value print_interf ?input_file ?output_file ast =
    sig_item_printer.val ?input_file ?output_file ast;
  value print_implem ?input_file ?output_file ast =
    str_item_printer.val ?input_file ?output_file ast;
end;

value enable_ocaml_printer () =
  let module M = OCamlPrinter PP.OCaml.Id PP.OCaml.MakeMore in ();

value enable_ocamlr_printer () =
  let module M = OCamlPrinter PP.OCamlr.Id PP.OCamlr.MakeMore in ();

(* value enable_ocamlrr_printer () =
  let module M = OCamlPrinter PP.OCamlrr.Id PP.OCamlrr.MakeMore in ();    *)

value enable_dump_ocaml_ast_printer () =
  let module M = OCamlPrinter PP.DumpOCamlAst.Id PP.DumpOCamlAst.Make in ();

value enable_dump_camlp4_ast_printer () =
  let module M = Printer PP.DumpCamlp4Ast.Id PP.DumpCamlp4Ast.Make in ();

value enable_null_printer () =
  let module M = Printer PP.Null.Id PP.Null.Make in ();
```

Listing 33: Camlp4 Register

Notice that functors Plugin, SyntaxExtension, OCamlSyntaxExtension, OCamlSyntaxExtension, SyntaxPlugin, they did the same thing essentially, they apply the second Funtor to Syntax(Camlp4.PreCast.Syntax).

Functors Printer, OCamlPrinter, OCamlPrinter, they did the same thing, apply the Make to Syntax, then register it.

Functors Parser, OCamlParser, did the same thing.

Functors AstFilter did nothing interesting.

It sticks to the toplevel Listing

Listing 34: Camlp4 Register Part 2

It mainly hook some global variables, like `Camlp4.Register.loaded_modlules`, but there's no fresh meat in this file. To conclude, Register did nothing, except making your code more modular, or register your syntax extension.

As we said, 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"]}
```

4.4 Camlp4Ast

As the code Listing 35 demonstrate below, there are several categories including *ident, ctyp,patt,expr, module_type, sig_item, with_constr, binding, rec_binding, module_binding, match_case, module_expr,str_item, class_type, class_sig_item, class_expr, class_str_item,*. And there are antiquotations for each syntax category, i.e, *IdAnt,TyAnt,PaAnt,ExAnt,MtAnt, SgAnt, WcAnt, BiAnt, RbAnt,MbAnt,McAnt,MeAnt,StAnt,CtAnt,CgAnt, CeAnt, CrAnt*

```
(** CAMLP4AST RevisedSyntax *)
type loc = Loc.t
and meta_bool =
  [ BTrue
  | BFalse
  | BAnt of string ]
and rec_flag =
  [ ReRecursive
```

```
  | ReNil
  | ReAnt of string ]
and direction_flag =
  [ DiTo
  | DiDownto
  | DiAnt of string ]
and mutable_flag =
  [ MuMutable
  | MuNil
  | MuAnt of string ]
and private_flag =
  [ PrPrivate
  | PrNil
  | PrAnt of string ]
and virtual_flag =
  [ ViVirtual
  | ViNil
  | ViAnt of string ]
and override_flag =
  [ OvOverride
  | OvNil
  | OvAnt of string ]
and row_var_flag =
  [ RvRowVar
  | RvNil
  | RvAnt of string ]
and meta_option 'a =
  [ ONone
  | OSome of 'a
  | OAnt of string ]
and meta_list 'a =
  [ LNil
  | LCons of 'a and meta_list 'a
  | LAnt of string ]
```

```
and ident =
  [ IdAcc of loc and ident and ident (* i . i *)
(*  <:ident< a . b >>   Access in module
   IdAcc of Loc.t and ident and ident *)
  | IdApp of loc and ident and ident (* i i *)
(*  <:ident< a b >>
   Application
```

```
   IdApp of Loc.t and ident and ident ??? *)
  | IdLid of loc and string (* foo *)
(* <:ident< $lid:i$ >>
   Lowercase identifier
   IdLid of Loc.t and string
*)
  | IdUid of loc and string (* Bar *)
(* <:ident< $uid:i$ >>
   Uppercase identifier
   IdUid of Loc.t and string
*)
  | IdAnt of loc and string (* $s$ *)
(* <:ident< $anti:s$ >>
   Antiquotation
   IdAnt of Loc.t and string
*)
 ]
(*  <:ident< $list:x$ >>
   list of accesses
   Ast.idAcc_of_list x  use IdAcc to accumulate to a list
 *)
and ctyp =
  [ TyNil of loc
  (*<:ctyp< >>  Empty typeTyNil of Loc.t *)
  | TyAli of loc and ctyp and ctyp (* t as t *) (* list 'a as 'a *)
  (* <:ctyp< t as t >>  Type aliasing
     TyAli of Loc.t and ctyp and ctyp *)
  | TyAny of loc (* _ *)
  (*  <:ctyp< _ >> Wildcard TyAny of Loc.t *)
  | TyApp of loc and ctyp and ctyp (* t t *) (* list 'a *)
  (* <:ctyp< t t >> Application TyApp of Loc.t and ctyp and ctyp *)
  | TyArr of loc and ctyp and ctyp (* t -> t *) (* int -> string *)
  (* <:ctyp< t -> t >>Arrow TyArr of Loc.t and ctyp and ctyp       *)
  | TyCls of loc and ident (* #i *) (* #point *)
  (* <:ctyp< #i >> Class type TyCls of Loc.t and ident     ??? *)
  | TyLab of loc and string and ctyp (* ~s:t *)
  (* <:ctyp< ~s >>  Label type TyLab of Loc.t and string and ctyp    *)
  | TyId  of loc and ident (* i *) (* Lazy.t *)
  (* <:ctyp< $id:i$ >>  Type identifier of TyId of Loc.t and ident *)
  (* <:ctyp< $lid:i$ >> TyId (_, IdLid (_, i)) *)
  (* <:ctyp< $uid:i$ >>  TyId (_, IdUid (_, i)) *)
  | TyMan of loc and ctyp and ctyp (* t == t *) (* type t = [ A | B ] == Foo.t *)
  (* <:ctyp< t == t >>  Type manifest TyMan of Loc.t and ctyp and ctyp       *)

  (* type t 'a 'b 'c = t constraint t = t constraint t = t *)
  | TyDcl of loc and string and list ctyp and ctyp and list (ctyp * ctyp)
  (* <:ctyp< type t 'a 'b 'c = t constraint t = t constraint t = t >>
```

```
   Type declaration
   TyDcl of Loc.t and string and list ctyp and ctyp and list (ctyp * ctyp) *)

(* < (t)? (..)? > *) (* < move : int -> 'a .. > as 'a  *)
| TyObj of loc and ctyp and row_var_flag
(* <:ctyp< < (t)? (..)? > >> Object type TyObj of Loc.t and ctyp and meta_bool       *)

| TyOlb of loc and string and ctyp (* ?s:t *)
(* <:ctyp< ?s:t >>  Optional label type TyOlb of Loc.t and string and ctyp       *)
| TyPol of loc and ctyp and ctyp (* ! t . t *) (* ! 'a . list 'a -> 'a *)
(* <:ctyp< ! t . t >> = Polymorphic type TyPol of Loc.t and ctyp and ctyp       *)
| TyQuo of loc and string (* 's *)
(* <:ctyp< 's >>' TyQuo of Loc.t and string       *)
| TyQuP of loc and string (* +'s *)
(* <:ctyp< +'s >> TyQuP of Loc.t and string       *)
| TyQuM of loc and string (* -'s *)
(* <:ctyp< -'s >> TyQuM of Loc.t and string       *)
| TyVrn of loc and string (* `s *)
(* <:ctyp< `s >> Polymorphic variant of TyVrn of Loc.t and string       *)
| TyRec of loc and ctyp (* { t } *) (* { foo : int ; bar : mutable string } *)
(* <:ctyp< { t } >>  Record TyRec of Loc.t and ctyp *)

| TyCol of loc and ctyp and ctyp (* t : t *)
(* <:ctyp< t : t >>Field declarationTyCol of Loc.t and ctyp and ctyp       *)
| TySem of loc and ctyp and ctyp (* t; t *)
(* <:ctyp< t; t >>Semicolon-separated type listTySem of Loc.t and ctyp and ctyp       *)
| TyCom of loc and ctyp and ctyp (* t, t *)
(* <:ctyp< t, t >>Comma-separated type listTyCom of Loc.t and ctyp and ctyp *)
| TySum of loc and ctyp (* [ t ] *) (* [ A of int and string | B ] *)
(* <:ctyp< [ t ] >>Sum typeTySum of Loc.t and ctyp       *)
| TyOf  of loc and ctyp and ctyp (* t of t *) (* A of int *)
(* <:ctyp< t of t >>TyOf of Loc.t and ctyp and ctyp       *)
| TyAnd of loc and ctyp and ctyp (* t and t *)
(* <:ctyp< t and t >>TyAnd of Loc.t and ctyp and ctyp *)
| TyOr  of loc and ctyp and ctyp (* t | t *)
(* <:ctyp< t | t >>"Or" pattern between typesTyOr of Loc.t and ctyp and ctyp       *)
| TyPrv of loc and ctyp (* private t *)
(* <:ctyp< private t >>Private type TyPrv of Loc.t and ctyp       *)
| TyMut of loc and ctyp (* mutable t *)
(* <:ctyp< mutable t >>  Mutable type TyMut of Loc.t and ctyp       *)
| TyTup of loc and ctyp (* ( t ) *) (* (int * string) *)
(* <:ctyp< ( t ) >> or <:ctyp< $tup:t$ >>  Tuple of TyTup of Loc.t and ctyp       *)
| TySta of loc and ctyp and ctyp (* t * t *)
(* <:ctyp< t * t >> TySta of Loc.t and ctyp and ctyp       *)
| TyVrnEq of loc and ctyp (* [ = t ] *)
(* <:ctyp< [ = t ] >>  TyVrnEq of Loc.t and ctyp       *)
| TyVrnSup of loc and ctyp (* [ > t ] *)
```

```
  (* <:ctyp< [ > t ] >> open polymorphic variant type TyVrnSup of Loc.t and ctyp       *)
  | TyVrnInf of loc and ctyp (* [ < t ] *)
  (* <:ctyp< [ < t ] >>  closed polymorphic variant type with no known tags
      TyVrnInf of Loc.t and ctyp       *)
  | TyVrnInfSup of loc and ctyp and ctyp (* [ < t > t ] *)
  (*  <:ctyp< [ < t > t ] >> closed polymorphic variant type with some known tags
      TyVrnInfSup of Loc.t and ctyp and ctyp
  *)
  | TyAmp of loc and ctyp and ctyp (* t & t *)
  (* <:ctyp< t & t >>conjuntive type in polymorphic variants
     TyAmp of Loc.t and ctyp and ctyp   *)
  | TyOfAmp of loc and ctyp and ctyp (* t of & t *)
  (* <:ctyp< $t1$ of & $t2$ >>Special (impossible) constructor (t1)
     that has both no arguments and arguments compatible with t2 at the
     same time.TyOfAmp of Loc.t and ctyp and ctyp *)
  | TyPkg of loc and module_type (* (module S) *)
  (* <:ctyp<(module S) >>  TyPkg of loc and module_type    *)
  | TyAnt of loc and string (* $s$ *)
  (* <:ctyp< $anti:s$ >>AntiquotationTyAnt of Loc.t and string       *)
  (*<:ctyp< $list:x$ >>list of accumulated ctyps
  depending on context,
  Ast.tyAnd_of_list,
  Ast.tySem_of_list,
  Ast.tySta_of_list,
  Ast.tyOr_of_list,
  Ast.tyCom_of_list,
  Ast.tyAmp_of_list

  In a closed variant type <:ctyp< [ < $t1$ > $t2$ ] >> the type t2 must
    not be the empty type; use a TyVrnInf node in this case.

  Type conjuctions are stored in a TyAmp tree, use Camlp4Ast.list_of_ctyp and
  Camlp4Ast.tyAmp_of_list to convert from and to a list of types.

  Variant constructors with arguments and polymorphic variant
  constructors with arguments are both represented with a TyOf
  node. For variant types the first TyOf type is an uppercase
  identifier (TyId), for polymorphic variant types it is an TyVrn
  node.

  Constant variant constructors are simply represented as
  uppercase identifiers (TyId). Constant polymorphic variant
  constructors take a TyVrn node.  *)
]
and patt =
  [ PaNil of loc
  (* <:patt< >>Empty patternPaNil of Loc.t *)
```

```
| PaId  of loc and ident (* i *)
(* <:patt< $id:i$ >>IdentifierPaId of Loc.t and ident       *)
(* <:patt< $lid:i$ >>PaId (_, IdLid (_, i)) *)
(* <:patt< $uid:i$ >>PaId (_, IdUid (_, i)) *)
| PaAli of loc and patt and patt (* p as p *) (* (Node x y as n) *)
(* <:patt< ( p as p ) >>AliasPaAli of Loc.t and patt and patt       *)
| PaAnt of loc and string (* $s$ *)
(* <:patt< $anti:s$ >>AntiquotationPaAnt of Loc.t and string       *)
| PaAny of loc (* _ *)
(* <:patt< _ >>WildcardPaAny of Loc.t       *)
| PaApp of loc and patt and patt (* p p *) (* fun x y -> *)
(* <:patt< p p >>ApplicationPaApp of Loc.t and patt and patt       *)
| PaArr of loc and patt (* [| p |] *)
(* <:patt< [| p |] >>ArrayPaArr of Loc.t and patt       *)
| PaCom of loc and patt and patt (* p, p *)
(* <:patt< p, p >>Comma-separated pattern listPaCom of Loc.t and patt and patt       *)
| PaSem of loc and patt and patt (* p; p *)
(* <:patt< p; p >>Semicolon-separated pattern listPaSem of Loc.t and patt and patt       *)
| PaChr of loc and string (* c *) (* 'x' *)
(* <:patt< $chr:c$ >>CharacterPaChr of Loc.t and string       *)
| PaInt of loc and string
(* <:patt< $int:i$ >>IntegerPaInt of Loc.t and string       *)
| PaInt32 of loc and string
(* <:patt< $int32:i$ >>Int32PaInt32 of Loc.t and string       *)
| PaInt64 of loc and string
(* <:patt< $int64:i$ >>Int64PaInt64 of Loc.t and string       *)
| PaNativeInt of loc and string
(* <:patt< $nativeint:i$ >>NativeIntPaNativeInt of Loc.t and string       *)
| PaFlo of loc and string
(* <:patt< $flo:f$ >>FloatPaFlo of Loc.t and string       *)
| PaLab of loc and string and patt (* ~s or ~s:(p) *)
(* <:patt< ~s >> <:patt< s:(p) >>LabelPaLab of Loc.t and string and patt       *)

(* ?s or ?s:(p) *)
| PaOlb of loc and string and patt
(* <:patt< ?s >> <:patt< ?s:(p) >>Optional labelPaOlb of Loc.t and string and patt       *)

(* ?s:(p = e) or ?(p = e) *)
| PaOlbi of loc and string and patt and expr
(* <:patt< ?s:(p = e) >> <:patt< ?(p = e) >
  >Optional label with default valuePaOlbi of Loc.t and string and patt and expr       *)

| PaOrp of loc and patt and patt (* p | p *)
(* <:patt< p | p >>OrPaOrp of Loc.t and patt and patt       *)
| PaRng of loc and patt and patt (* p .. p *)
(* <:patt< p .. p >>Pattern rangePaRng of Loc.t and patt and patt *)
| PaRec of loc and patt (* { p } *)
```

```
  (* <:patt< { p } >>RecordPaRec of Loc.t and patt *)
  | PaEq  of loc and ident and patt (* i = p *)
  (* <:patt< i = p >>EqualityPaEq of Loc.t and ident and patt       *)
  | PaStr of loc and string (* s *)
  (* <:patt< $str:s$ >>StringPaStr of Loc.t and string *)
  | PaTup of loc and patt (* ( p ) *)
  (* <:patt< ( $tup:p$ ) >>TuplePaTup of Loc.t and patt       *)
  | PaTyc of loc and patt and ctyp (* (p : t) *)
  (* <:patt< (p : t) >>Type constraintPaTyc of Loc.t and patt and ctyp       *)
  | PaTyp of loc and ident (* #i *)
  (* <:patt< #i >>PaTyp of Loc.t and ident
     used in polymorphic variants
  *)
  | PaVrn of loc and string (* `s *)
  (* <:patt< `s >>Polymorphic variantPaVrn of Loc.t and string   *)
  | PaLaz of loc and patt (* lazy p *)
  (* <:patt< lazy x >> *)
 ]
 (* <:patt< $list:x$ >>list of accumulated patts depending on context,
    Ast.paCom_of_list, Ast.paSem_of_list Tuple elements are wrapped in a
    PaCom tree. The utility functions Camlp4Ast.paCom_of_list and
    Camlp4Ast.list_of_patt convert from and to a list of tuple
    elements.  *)
and expr =
  [ ExNil of loc
      (* <:expr< >> *)
  | ExId  of loc and ident (* i *)
      (* <:expr< $id:i$ >> notice that antiquot id requires ident directly *)
      (* <:expr< $lid:i$ >> ExId(_,IdLid(_,i)) *)
      (* <:expr< $uid:i$ >> ExId(_, IdUid(_,i)) *)
  | ExAcc of loc and expr and expr (* e.e *)
      (* <:expr< $e1$.$e2$ >> Access in module ? *)
  | ExAnt of loc and string (* $s$ *)
      (* <:expr< $anti:s$ >> *)
  | ExApp of loc and expr and expr (* e e *)
      (* <:expr< $e1$ $e2$ >> Application *)
  | ExAre of loc and expr and expr (* e.(e) *)
      (* <:expr< $e$.($e$) >> Array access  *)
  | ExArr of loc and expr (* [| e |] *)
      (* <:expr< [|$e$| ] Array declaration *)
  | ExSem of loc and expr and expr (* e; e *)
      (* <:expr< $e$; $e$ >>  *)
  | ExAsf of loc (* assert False *)
      (* <:expr< assert False >> *)
  | ExAsr of loc and expr (* assert e *)
      (* <:expr< assert $e$ >>  *)
  | ExAss of loc and expr and expr (* e := e *)
```

```
    (* <:expr< $e$ := $e$ >>  *)
| ExChr of loc and string (* 'c' *)
    (* <:exp< $`chr:s$ >> Character *)
| ExCoe of loc and expr and ctyp and ctyp (* (e : t) or (e : t :> t) *)
    (* <:expr< ($e$:> $t$) >> <:expr< ($e$ : $t1$ :> $t2$ ) >>
       The first ctyp is populated by TyNil
    *)
| ExFlo of loc and string (* 3.14 *)
    (* <:expr< $flo:f$ >> *)
    (* <:expr< $`flo:f$ >>  ExFlo(_,string_of_float f) *)

(* for s = e to/downto e do { e } *)
| ExFor of loc and string and expr and expr and direction_flag and expr
    (* <:expr< for $s$ = $e1$ to/downto $e2$ do { $e$  } >>  *)
| ExFun of loc and match_case (* fun [ mc ] *)
    (* <:expr< fun [ $a$ ] >>  *)
| ExIfe of loc and expr and expr and expr (* if e then e else e *)
    (* <:expr< if $e$ then $e$ else $e$ >>  *)
| ExInt of loc and string (* 42 *)
    (* <:expr< $int:i$ >> *)
    (* <:expr< $`int:i$ >> ExInt(_, string_of_int i) *)
| ExInt32 of loc and string
    (* <:expr< $int32:i$ >>
       <:expr< $`int32:i$ >>
    *)
| ExInt64 of loc and string
    (* <:expr< $int64:i$ >> *)
    (* <:expr< $`int64:i$ >>  *)
| ExNativeInt of loc and string
    (* <:expr< $nativeint:i$ >> <:expr< $`nativeint:i$ >>  *)
| ExLab of loc and string and expr (* ~s or ~s:e *)
    (* <:expr< ~ $s$ >> ExLab (_, s, ExNil) *)
    (* <:expr< ~ $s$ : $e$ >>  *)
| ExLaz of loc and expr (* lazy e *)
    (* <:expr< lazy $e$ >>  *)


| ExLet of loc and rec_flag and binding and expr
    (* <:expr< let $b$ in $e$ >> *)
    (* <:expr< let rec $b$ in $e$ >>    *)

| ExLmd of loc and string and module_expr and expr
    (* <:expr< let module $s$ = $me$ in $e$ >>  *)


| ExMat of loc and expr and match_case
    (* <:expr< match $e$ with [ $a$ ] >>  *)
```

```
(* new i *)
| ExNew of loc and ident
    (* <:expr< new $id:i$ >>  new object *)
    (* <:expr< new $lid:str$ >>  *)

(* object ((p))? (cst)? end *)
| ExObj of loc and patt and class_str_item
    (* <:expr< object ( ($p$))? ($cst$)? end >>  object declaration *)

(* ?s or ?s:e *)
| ExOlb of loc and string and expr
    (* <:expr< ? $s$ >>      Optional label  *)
    (* <:expr< ? $s$ : $e$ >>  *)

| ExOvr of loc and rec_binding
(* <:expr< {< $rb$  >} >>  *)

| ExRec of loc and rec_binding and expr
    (* <:expr< { $b$ } >> *)
    (* <:expr< {($e$ ) with $b$ } >>  *)
| ExSeq of loc and expr
    (* <:expr< do { $e$ } >> *)
    (* <:expr< $seq:e$ >> *)
    (* another way to help you figure out the type *)
    (* type let f e = <:expr< $seq:e$ >> in the toplevel            *)

| ExSnd of loc and expr and string
    (* <:expr< $e$ # $s$ >> METHOD call  *)

| ExSte of loc and expr and expr
    (* <:expr< $e$.[$e$] >> String access  *)

| ExStr of loc and string
    (* <:expr< $str:s$ >> "\n" -> "\n" *)
    (* <:expr< $`str:s$ >> "\n" -> "\\n" *)

| ExTry of loc and expr and match_case
    (* <:expr< try $e$ with [ $a$ ] >>  *)


| ExTup of loc and expr
    (* <:expr< ( $tup:e$ ) >>  *)

| ExCom of loc and expr and expr
    (* <:expr< $e$, $e$ >>  *)
```

```
(* (e : t) *)
| ExTyc of loc and expr and ctyp
    (* <:expr< ($e$ : $t$ ) Type constraint  *)

| ExVrn of loc and string
    (* <:expr< `$s$ >>  *)

| ExWhi of loc and expr and expr
    (* <:expr< while $e$ do { $e$ } >>  *)

| ExOpI of loc and ident and expr
    (* <:expr< let open $id:i$ in $e$ >>  *)

| ExFUN of loc and string and expr
    (* <:expr< fun (type $s$ ) -> $e$  >> *)
    (* let f x (type t) y z = e *)

| ExPkg of loc and module_expr
   (* (module ME : S) which is represented as (module (ME : S)) *)
]
and module_type =
  (**
     mt ::=
     | (* empty *)
     | ident
     | functor (s : mt) -> mt
     | 's
     | sig sg end
     | mt with wc
     | $s$
  *)
  [ MtNil of loc

  | MtId  of loc and ident
  (* i *)  (* A.B.C *)
  (* <:module_type< $id:ident$ >> named module type  *)

  | MtFun of loc and string and module_type and module_type
  (* <:module_type< functor ($uid:s$ : $mtyp:mta$ ) -> $mtyp:mtr$ >> *)

  | MtQuo of loc and string
  (* 's *)

  | MtSig of loc and sig_item
  (* sig sg end *)
  (* <:module_type< sig $sigi:sig_items$ end >>  *)
```

```
    | MtWit of loc and module_type and with_constr
    (* mt with wc *)
    (* <:module_type< $mtyp:mt$ with $with_constr:with_contr$ >>  *)
    | MtOf of loc and module_expr
    (* module type of m *)

    | MtAnt of loc and string
    (* $s$ *)
]
  (** Several with-constraints are stored in an WcAnd tree. Use
      Ast.wcAnd_of_list and Ast.list_of_with_constr to convert from and to a
      list of with-constraints.  Several signature items are stored in an
      SgSem tree. Use Ast.sgSem_of_list and Ast.list_of_sig_item to convert
      from and to a list of signature items. *)
and sig_item =
    (*
      sig_item, sg ::=
   | (* empty *)
   | class cict
   | class type cict
   | sg ; sg
   | #s
   | #s e
   | exception t
   | external s : t = s ... s
   | include mt
   | module s : mt
   | module rec mb
   | module type s = mt
   | open i
   | type t
   | value s : t
   | $s$
   lacking documentation !!
    *)
  [ SgNil of loc

    (* class cict *)
    (* <:sig_item< class $s$ >>;; *)
    (* <:sig_item< class $typ:s$ >>;; *)
    | SgCls of loc and class_type

    (* class type cict *)
    (* <:sig_item< class type $s$ >>;; *)
    (* <:sig_item< class type $typ:s$ >>;;  *)
    | SgClt of loc and class_type
```

```
(* sg ; sg *)
| SgSem of loc and sig_item and sig_item

(* # s or # s e ???  *)
(* Directive  *)
| SgDir of loc and string and expr

(* exception t *)
| SgExc of loc and ctyp

(* external s : t = s ... s *)
(*  <:sig_item< external $lid:id$ : $typ:type$ = $str_list:string_list$ >>
    another antiquot str_list
*)
| SgExt of loc and string and ctyp and meta_list string

(* include mt *)
| SgInc of loc and module_type

(* module s : mt *)
(* <:sig_item< module $uid:id$ : $mtyp:mod_type$ >>
   module Functor declaration
*)
(**
   <:sig_item< module $uid:mid$ ( $uid:arg$ : $mtyp:arg_type$ ) : $mtyp:res_type$ >>
   -->
   <:sig_item< module $uid:mid$ : functor ( $uid:arg$ : $mtyp:arg_type$ ) -> $mtyp:res_type$ >>
*)
| SgMod of loc and string and module_type

(* module rec mb *)
| SgRecMod of loc and module_binding

(* module type s = mt *)
(* <:sig_item< module type $uid:id$ = $mtyp:mod_type$ >>
   module type declaration
*)
(**
   <:sig_item< module type $uid:id$ >> abstract module type
   -->
   <:sig_item< module type $uid:id$ = $mtyp:<:module_type< >>$ >>
*)
| SgMty of loc and string and module_type

(* open i *)
| SgOpn of loc and ident
```

```
    (* type t *)
    (* <:sig_item< type $typ:type$ >>  *)
   (** <:sig_item< type $lid:id$ $p1$ ... $pn$ = $t$ constraint $c1l$
       = $c1r$ ... constraint $cnl$ = $cnr$ >>

       type declaration
       SgTyp
       of Loc.t and (TyDcl of Loc.t and id and [p1;...;pn] and t and
       [(c1l, c1r); ... (cnl, cnr)]) *)
   | SgTyp of loc and ctyp

    (* value s : t *)
    (* <:sig_item< value $lid:id$ : $typ:type$ >>  *)
    | SgVal of loc and string and ctyp

    | SgAnt of loc and string (* $s$ *) ]
  (** An exception is treated like a single type constructor. For
      exception declarations the type should be either a type
      identifier (TyId) or a type constructor (TyOf).

      Abstract module type declarations (i.e., module type
      declarations without equation) are represented with the empty
      module type.

      Mutually recursive type declarations (separated by
      and) are stored in a TyAnd tree. Use Ast.list_of_ctyp and
      Ast.tyAnd_of_list to convert to and from a list of type
      declarations.

      The quotation parser for types (<:ctyp< ... >>) does not parse
      type declarations. Type declarations must therefore be embedded
      in a sig_item or str_item quotation.

      There seems to be no antiquotation syntax for a list of type
      parameters and a list of constraints inside a type
      declaration. The existing form can only be used for a fixed
      number of type parameters and constraints.

      Complete class and class type declarations (including name and
      type parameters) are stored as class types.

      Several "and" separated class or class type declarations are
      stored in a CtAnd tree, use Ast.list_of_class_type and
      Ast.ctAnd_of_list to convert to and from a list of class
      types.  *)
and with_constr =
    (**
```

```
      with_constraint, with_constr, wc ::=
   | wc and wc
   | type t = t
   | module i = i
    *)
  [ WcNil of loc

    (* type t = t *)
    (* <:with_constr< type $typ:type_1$ = $typ:type_2$ >>  *)
    | WcTyp of loc and ctyp and ctyp

    (* module i = i *)
    (* <:with_constr< module $id:ident_1$ = $id:ident_2$ >>  *)
    | WcMod of loc and ident and ident

    (* type t := t *)
    | WcTyS of loc and ctyp and ctyp

    (* module i := i *)
    | WcMoS of loc and ident and ident

    (* wc and wc *)
    (* <:with_constr< $with_constr:wc1$ and $with_constr:wc2$ >> *)
    | WcAnd of loc and with_constr and with_constr

    | WcAnt of loc and string (* $s$ *)
]
  (** Several with-constraints are stored in an WcAnd tree. Use
      Ast.wcAnd_of_list and Ast.list_of_with_constr to convert from and
      to a list of with-constraints. *)
and binding =
  (**  binding, bi ::=
   | bi and bi
   | p = e *)
  [ BiNil of loc

    (* bi and bi *)  (* let a = 42 and c = 43 *)
    (* <:binding< $b1$ and $b2$ >>  *)
    | BiAnd of loc and binding and binding

    (* p = e *)  (* let patt = expr *)
    (* <:binding< $pat:pattern$ = $exp:expression$ >>
       <:binding< $p$ = $e$ >> ;; both are ok
    *)
    (**
       <:binding< $pat:f$ $pat:x$ = $exp:ex$ >>
       -->
```

```
        <:binding< $pat:f$ = fun $pat:x$ -> $exp:ex$ >>

        <:binding< $pat:p$ : $typ:type$ = $exp:ex$ >>
        typed binding -->
        <:binding< $pat:p$ = ( $exp:ex$ : $typ:type$ ) >>

        <:binding< $pat:p$ :> $typ:type$ = $exp:ex$ >>
        coercion binding -->
        <:binding< $pat:p$ = ( $exp:ex$ :> $typ:type$ ) >>
    *)
    | BiEq  of loc and patt and expr

    | BiAnt of loc and string (* $s$ *)
    (**
        The utility functions Camlp4Ast.biAnd_of_list and
        Camlp4Ast.list_of_bindings convert between the BiAnd tree of
        parallel bindings and a list of bindings.  The utility
        functions Camlp4Ast.binding_of_pel and pel_of_binding convert
        between the BiAnd tree and a pattern * expression lis
    *)  ]
and rec_binding =
    (** record bindings
         record_binding, rec_binding, rb ::=
         | rb ; rb
         | x = e
    *)
  [ RbNil of loc

    (* rb ; rb *)
    | RbSem of loc and rec_binding and rec_binding

    (* i = e
        very simple
    *)
    | RbEq  of loc and ident and expr

    | RbAnt of loc and string (* $s$ *)
    ]

  and module_binding =
    (**
        Recursive module bindings
        module_binding, mb ::=
        | (* empty *)
        | mb and mb
        | s : mt = me
        | s : mt
```

```
      | $s$
    *)
    [ MbNil of loc

    (* mb and mb *) (* module rec (s : mt) = me and (s : mt) = me *)
    | MbAnd of loc and module_binding and module_binding

    (* s : mt = me *)
    | MbColEq  of loc and string and module_type and module_expr

    (* s : mt *)
    | MbCol  of loc and string and module_type

    | MbAnt of loc and string (* $s$ *) ]

and match_case =
    (**
       match_case, mc ::=
       | (* empty *)
       | mc | mc
       | p when e -> e
       | p -> e
    (* a sugar for << p when e1 -> e2 >> where e1 is the empty expression *)
    <:match_case< $list:mcs$ >>list of or-separated match casesAst.mcOr_of_list
    *)
    [
      (* <:match_case< >>  *)
      McNil of loc

    (* a | a *)
    (* <:match_case< $mc1$ | $mc2$ >>            *)
    | McOr of loc and match_case and match_case

    (* p (when e)? -> e *)
    (* <:match_case< $p$ -> $e$ >>    *)
    (* <:match_case< $p$ when $e1$ or $e2$ >>    *)
    | McArr of loc and patt and expr and expr

    (* <:match_case< $anti:s$ >>    *)
    | McAnt of loc and string (* $s$ *) ]

and module_expr =
    (**
        module_expression, module_expr, me ::=
       | (* empty *)
       | ident
       | me me
```

```
    | functor (s : mt) -> me
    | struct st end
    | (me : mt)
    | $s$
    | (value pexpr : ptype)
  *)
  [
    (* <:module_expr< >>  *)
    MeNil of loc

  (* i *)
  (* <:module_expr< $id:mod_ident$ >>  *)
  | MeId  of loc and ident

  (* me me *)
  (* <:module_expr< $mexp:me$ $mexp:me$ >>         Functor application *)
  | MeApp of loc and module_expr and module_expr

  (* functor (s : mt) -> me *)
  (* <:module_expr< functor ($uid:id$ : $mtyp:mod_type$) -> $mexp:me$ >>  *)
  | MeFun of loc and string and module_type and module_expr

  (* struct st end *)
 (* <:module_expr< struct $stri:str_item$ end >>  *)
  | MeStr of loc and str_item

  (* (me : mt) *)
  (* <:module_expr< ($mexp:me$ : $mtyp:mod_type$ ) >>
     signature constraint
  *)
  | MeTyc of loc and module_expr and module_type

  (* (value e) *)
  (* (value e : S) which is represented as (value (e : S)) *)
  (* <:module_expr< (value $exp:expression$ ) >>
     module extraction

     <:module_expr< (value $exp:expression$ : $mtyp:mod_type$ ) >>
     -->
     <:module_expr<
     ( value $exp: <:expr< ($exp:expression$ : (module $mtyp:mod_type$ ) ) >> $ )
     >>
  *)
  | MePkg of loc and expr

  (* <:module_expr< $anti:string$ >>          *)
  | MeAnt of loc and string (* $s$ *)
```

```
(** Inside a structure several structure items are packed into a StSem
    tree. Use Camlp4Ast.stSem_of_list and Camlp4Ast.list_of_str_item to
    convert from and to a list of structure items.

    The expression in a module extraction (MePkg) must be a type
    constraint with a package type. Internally the syntactic class of
    module types is used for package types.
*)
]
  and str_item =
    (**
        structure_item, str_item, st ::=
   | (* empty *)
   | class cice
   | class type cict
   | st ; st
   | #s
   | #s e
   | exception t or exception t = i
   | e
   | external s : t = s ... s
   | include me
   | module s = me
   | module rec mb
   | module type s = mt
   | open i
   | type t
   | value b or value rec bi
   | $s$
    *)
    [ StNil of loc

    (* class cice *)
    (* <:str_item< class $cdcl:class_expr$ >>  *)
    | StCls of loc and class_expr

    (* class type cict *)
    (**
      <:str_item< class type $typ:class_type$ >>
       --> class type definition
    *)
    | StClt of loc and class_type

    (* st ; st *)
    (* <:str_item< $str_item_1$; $str_item_2$ >>  *)
    | StSem of loc and str_item and str_item
```

```
(* # s or # s e *)
(* <:str_item< # $string$ $expr$ >>  *)
| StDir of loc and string and expr

(* exception t or exception t = i *)
(* <:str_item< exception $typ:type$ >>   -> None *)
(* <:str_item< exception $typ:type$ >> ->
   Exception alias -> Some ident
*)
| StExc of loc and ctyp and meta_option(*FIXME*) ident

(* e *)
(* <:str_item< $exp:expr$ >> toplevel expression          *)
| StExp of loc and expr

(* external s : t = s ... s *)
(* <:str_item< external $lid:id$ : $typ:type$ = $str_list:string_list$ >>  *)
| StExt of loc and string and ctyp and meta_list string

(* include me *)
(* <:str_item< include $mexp:mod_expr$ >>  *)
| StInc of loc and module_expr

(* module s = me *)
(* <:str_item< module $uid:id$ = $mexp:mod_expr$ >>  *)
| StMod of loc and string and module_expr

(* module rec mb *)
(* <:str_item< module rec $module_binding:module_binding$ >>  *)
| StRecMod of loc and module_binding

(* module type s = mt *)
(* <:str_item< module type $uid:id$ = $mtyp:mod_type$ >> *)
| StMty of loc and string and module_type

(* open i *)
(* <:str_item< open $id:ident$ >>  *)
| StOpn of loc and ident

(* type t *)
(* <:str_item< type $typ:type$ >>  *)
(**
   <:str_item< type $lid:id$ $p1$ ... $pn$ = $t$
   constraint $c1l$ = $c1r$ ... constraint $cnl$ = $cnr$ >>
   -->
   StTyp of Loc.t and
```

```
      (TyDcl of Loc.t and id and [p1;...;pn] and t and [(c1l, c1r); ... (cnl, cnr)])
    *)
    | StTyp of loc and ctyp

    (* value (rec)? bi *)
    (* <:str_item< value $rec:r$ $binding$ >>          *)
    (* <:str_item< value rec $binding$ >> *)
    (* <:str_item< value  $binding$ >> *)
    | StVal of loc and rec_flag and binding

    (* <:str_item< $anti:s$ >>          *)
    | StAnt of loc and string (* $s$ *)
(**
  <:str_item< module $uid:id$ ( $uid:p$ : $mtyp:mod_type$ ) = $mexp:mod_expr$ >>
  ---->
  <:str_item< module $uid:id$ =
  functor ( $uid:p$ : $mtyp:mod_type$ ) -> $mexp:mod_expr$ >>

  <:str_item< module $uid:id$ : $mtyp:mod_type = $mexp:mod_expr$ >>
  ---->
  <:str_item< module $uid:id$ = ($mexp:mod_expr$ : $mtyp:mod_type ) >>

  <:str_item< type t >>
  ---->
  <:str_item< type t = $<:ctyp< >>$ >>

  <:str_item< # $id$ >> (directive without arguments)
  ---->
  <:str_item< # $a$ $<:expr< >>$ >>


  A whole compilation unit or the contents of a structure is given as
  *one* structure item in the form of a StSem tree.

  The utility functions Camlp4Ast.stSem_of_list and
  Camlp4Ast.list_of_str_item convert from and to a list of structure
  items.

  An exception is treated like a single type constructor. For
  exception definitions the type should be either a type identifier
  (TyId) or a type constructor (TyOf). For execption aliases it
  should only be a type identifier (TyId).

  Abstract types are represented with the empty type.

  Mutually recursive type definitions (separated by and) are stored
  in a TyAnd tree. Use Ast.list_of_ctyp and Ast.tyAnd_of_list to
```

```
    convert to and from a list of type definitions.

    The quotation parser for types (<:ctyp< ... >>) does not parse type
    declarations. Type definitions must therefore be embedded in a
    sig_item or str_item quotation.

    There seems to be no antiquotation syntax for a list of type
    parameters and a list of constraints inside a type definition. The
    existing form can only be used for a fixed number of type
    parameters and constraints.

    Complete class type definitions (including name and type
    parameters) are stored as class types.

    Several "and" separated class type definitions are stored in a
    CtAnd tree, use Ast.list_of_class_type and Ast.ctAnd_of_list to
    convert to and from a list of class types.

    Several "and" separated classes are stored in a CeAnd tree, use
    Ast.list_of_class_exprand Ast.ceAnd_of_list to convert to and from
    a list of class expressions.

    Several "and" separated recursive modules are stored in a MbAnd
    tree, use Ast.list_of_module_binding and Ast.mbAnd_of_list to
    convert to and from a list of module bindings.

    Directives without argument are represented with the empty
    expression argument.  *)
]
and class_type =
  (** Besides class types, ast nodes of this type are used to
      describe *class type definitions*
      (in structures and signatures)
      and class declarations (in signatures).

      class_type, ct ::=
   | (* empty *)
   | (virtual)? i ([ t ])?
   | [t] -> ct
   | object (t) csg end
   | ct and ct
   | ct : ct
   | ct = ct
   | $s$
    *)
  [ CtNil of loc
```

```
(* (virtual)? i ([ t ])? *)
(* <:class_type< $virtual:v$ $id:ident$ [$list:p$ ] >> *)
(* instanciated class type/ left hand side of a class *)
(* declaration or class type definition/declaration  *)
| CtCon of loc and virtual_flag and ident and ctyp

(* [t] -> ct *)
(* <:class_type< [$typ:type$] -> $ctyp:ct$ >>
     class type valued function
*)
| CtFun of loc and ctyp and class_type

(* object ((t))? (csg)? end *)
(* <:class_type< object ($typ:self_type$) $csg:class_sig_item$ end >> *)
(* class body type *)
| CtSig of loc and ctyp and class_sig_item

(* ct and ct *)
(* <:class_type< $ct1$ and $ct2$ >> *)
(* mutually recursive class types *)
| CtAnd of loc and class_type and class_type

(* ct : ct *)
(* <:class_type< $decl$ : $ctyp:ct$ >> *)
(* class  c : object .. end   class declaration as in
   "class c: object .. end " in a signature
*)
| CtCol of loc and class_type and class_type

(* ct = ct *)
(* <:class_type< $decl$ = $ctyp:ct$ >> *)
(* class type declaration/definition as in "class type c = object .. end " *)
| CtEq  of loc and class_type and class_type


(* $s$ *)
| CtAnt of loc and string
(**
   <:class_type< $id:i$ [ $list:p$] >>
   --->
   <:class_type< $virtual:Ast.BFalse$ $id:i$ [ $list:p$] >>

   <:class_type< $virtual:v$ $id:i$ >>
   --->
   <:class_type< $virtual:v$ $id:i$ [ $<:ctyp< >>$ ] >>

   <:class_type< object $x$ end >>
```

```
      --->
      <:class_type< object ($<:ctyp< >>$) $x$ end >>
    *)
(** CtCon is used for possibly instanciated/parametrized class
    type identifiers. They appear on the left hand side of class
    declaration and class definitions or as reference to existing
    class types. In the latter case the virtual flag is probably
    irrelevant.


    Several type parameters/arguments are stored in a TyCom tree, use
    Ast.list_of_ctyp and Ast.tyCom to convert to and from list of
    parameters/arguments.

    An empty type parameter list and an empty type argument is
    represented with the empty type.

    The self binding in class body types is represented by a type
    expression. If the self binding is absent, the empty type
    expression (<:ctype< >>) is used.

    Several class signature items are stored in a CgSem tree, use
    Ast.list_of_class_sig_item and Ast.cgSem_of_list to convert to and
    from a list of class signature items

*)
]
  and class_sig_item =
    (**
      class_signature_item, class_sig_item, csg ::=
   | (* empty *)
   | type t = t
   | csg ; csg
   | inherit ct
   | method s : t or method private s : t
   | value (virtual)? (mutable)? s : t
   | method virtual (mutable)? s : t
   | $s$
    *)
    [

      (* <:class_sig_item< >>  *)
      CgNil of loc

    (* <:class_sig_item< constraint $typ:type1$ = $typ:type2$ >>
      type constraint  *)
    | CgCtr of loc and ctyp and ctyp
```

```
    (* csg ; csg *)
    | CgSem of loc and class_sig_item and class_sig_item

    (* inherit ct *)
    (* <:classs_sig_item< inherit $ctyp:class_type$ >>  *)
    | CgInh of loc and class_type

    (* method s : t or method private s : t *)
    (* <:class_sig_item< method $private:pf$ $lid:id$:$typ:type$ >>    *)
    | CgMth of loc and string and private_flag and ctyp

    (* value (virtual)? (mutable)? s : t *)
    (* <:class_sig_item< value $mutable:mf$ $virtual:vf$ $lid:id$ : $typ:type$ >>  *)
    | CgVal of loc and string and mutable_flag and virtual_flag and ctyp

    (* method virtual (private)? s : t *)
    (* <:class_sig_item< method virtual $private:pf$ $lid:id$ : $typ:type$ >>   *)
    | CgVir of loc and string and private_flag and ctyp

    (* <:class_sig_item< $anti:a$ >>    *)
    | CgAnt of loc and string (* $s$ *)

    (**
       <:class_sig_item< type $typ:type_1$ = $typ:type_2$
       --->
       <:class_sig_item< constraint $typ:type_1$ = $typ:type_2$ >>

       The empty class signature item is used as a placehodler in
       empty class body types (class type e = object end )
    *)
]
and class_expr =
  (** Ast nodes of this type are additionally used to describe whole
      (mutually recursive) class definitions.


      class_expression, class_expr, ce ::=
   | (* empty *)
   | ce e
   | (virtual)? i ([ t ])?
   | fun p -> ce
   | let (rec)? bi in ce
   | object (p) (cst) end
   | ce : ct
   | ce and ce
   | ce = ce
```

```
  | $s$
 *)
 [
  CeNil of loc

  (* ce e *)
  (*
    <:class_expr< $cexp:ce$ $exp:exp$ >>

    application
  *)
  | CeApp of loc and class_expr and expr

  (* (virtual)? i ([ t ])? *)
  (* <:class_expr< $virtual:vf$ $id:ident$
     [ $typ:type_param$ ] >>

     instanciated class/ left hand side of class
     definitions.

     CeCon of Loct.t and vf and ident and type_param
  *)
  | CeCon of loc and virtual_flag and ident and ctyp

  (* fun p -> ce *)
  (* <:class_expr< fun $pat:pattern$ -> $cexp:ce$ >>
     class valued funcion

     CeFun of Loc.t and pattern and ce
  *)
  | CeFun of loc and patt and class_expr

  (* let (rec)? bi in ce *)
  (* <:class_expr< let $rec:rf$ $binding:binding$ in $cexp:ce$ >>  *)
  | CeLet of loc and rec_flag and binding and class_expr

  (* object ((p))? (cst)? end *)
  (* <:class_expr< object ( $pat:self_binding$ ) $cst:class_str_items$ end >> *)
  | CeStr of loc and patt and class_str_item

  (* ce : ct
     type constraint
     <:class_expr< ($cexp:ce$ : $ctyp:class_type$) >>
  *)
  | CeTyc of loc and class_expr and class_type

  (* ce and ce
```

```
    mutually recursive class definitions
  *)
  | CeAnd of loc and class_expr and class_expr

  (**
    <:class_expr< $ci$ = $cexp:ce$ >>
    class definition as in class ci = object .. end
  *)
  | CeEq  of loc and class_expr and class_expr

  (* $s$ *)
  (** <:class_expr< $anti:s$ >> *)
  | CeAnt of loc and string
  (**
    <:class_expr< $id:id$ [$tp$] >>
    ----> non-virtual class/ instanciated class
    <:class_expr< $virtual:Ast.BFalse$ $id:id$ [$tp$ ] >>

    <:class_expr< $virtual:vf$ $id:id$ >>
    ---->
    <:class_expr< $virtual:vf$ $id:id$ [ $<:ctyp< >>$ ] >>

    <:class_expr< fun $pat:p1$ $pat:p2$ -> $cexp:ce$ >>
    ---->
    <:class_expr< fun $pat:p1$ -> fun $pat:p2$ -> $cexp:ce$ >>

    <:class_expr< let $binding:bi$ in $cexp:ce$ >>
    ---->
    <:class_expr< let $rec:Ast.BFalse$ $binding:bi$ in $cexp:ce$ >>

    <:class_expr< let $rec:Ast.BFalse$ $binding:bi$ in $cexp:ce$ >>
    ---->
    <:class_expr< object ( $<:patt< >>$ ) $cst:cst$ end >>
  *)
(** No type parameters or arguments in an instanciated class
  (CeCon) are represented with the empty type (TyNil).

  Several type parameters or arguments in an instanciated class
  (CeCon) are stored in a TyCom tree. Use Ast.list_of_ctyp and
  Ast.tyCom_of_list convert to and from a list of type parameters.

  There are three common cases for the self binding in a class
  structure: An absent self binding is represented by the
  empty pattern (PaNil).  An identifier (PaId) binds the
  object.  A typed pattern (PaTyc) consisting of an identifier
  and a type variable binds the object and the self type.
```

```
    More than one class structure item are stored in a CrSem
    tree. Use Ast.list_of_class_str_item and Ast.crSem_of_list to
    convert to and from a list of class items.
*)
]
and class_str_item =
    (**
       class_structure_item, class_str_item, cst ::=
   | (* empty *)
   | cst ; cst
   | type t = t
   | inherit(!)? ce (as s)?
   | initializer e
   | method(!)? (private)? s : t = e or method (private)? s = e
   | value(!)? (mutable)? s = e
   | method virtual (private)? s : t
   | value virtual (private)? s : t
   | $s$
    *)
  [
    CrNil of loc

    (* cst ; cst *)
    | CrSem of loc and class_str_item and class_str_item

    (* type t = t *)
    (* <:class_str_item< constraint $typ:type_1$ = $typ:type_2$ >>
       type constraint
    *)
    | CrCtr of loc and ctyp and ctyp

    (* inherit(!)? ce (as s)? *)
    (* <:class_str_item< inherit $!:override$ $cexp:class_cexp$ as $lid:id$ >>  *)
    | CrInh of loc and override_flag and class_expr and string

    (* initializer e *)
    (* <:class_str_item< initializer $exp:expr$ >>  *)
    | CrIni of loc and expr

    (** method(!)? (private)? s : t = e or method(!)? (private)? s = e *)
    (** <:class_str_item< method $!override$ $private:pf$ $lid:id$: $typ:poly_type$ =
        $exp:expr$ >>
    *)
    | CrMth of loc and string and override_flag and private_flag and expr and ctyp

    (* value(!)? (mutable)? s = e *)
    (** <:class_str_item< value $!:override$ $mutable:mf$ $lid:id$ = $exp:expr$ >>
```

```
       instance variable
    *)
    | CrVal of loc and string and override_flag and mutable_flag and expr

    (** method virtual (private)? s : t *)
    (** <:class_str_item< method virtual $private:pf$ $lid:id$ : $typ:poly_type$ >>
       virtual method
    *)
    | CrVir of loc and string and private_flag and ctyp

    (* value virtual (mutable)? s : t *)
    (* <:class_str_item< value virtual $mutable:mf$ $lid:id$ : $typ:type$ >> *)
    (* virtual instance variable  *)
    | CrVvr of loc and string and mutable_flag and ctyp

    | CrAnt of loc and string (* $s$ *)

(**
   << constraint $typ:type_1$ = $typ:type_2$ >>
   ----> type constraint
   << type $typ:type1$ = $typ:type2$ >>

   << inherit $!:override$ $cexp:class_exp$ >>
   ---> superclass without binding
   << inherit $!:override$ $cexp:class_exp$ as $lid:""$ >>

   <<inherit $cexp:class_exp$ as $lid:id$ >>
   ---> superclass without override
   <<inherit $!:Ast.OvNil$ $cexp:class_exp$ as $lid:id$ >>

   <:class_str_item< method $private:pf$ $lid:id$ : $typ:poly_type$ = $exp:expr$ >>
   non-overriding method
   <:class_str_item< method $!:Ast.OvNil$
   $private:pf$ $lid:id$ : $typ:poly_type$ = $exp:expr$ >>

   <:class_str_item< method $private:pf$ $lid:id$ = $exp:expr$ >>
   monomorphic method
   <:class_str_item< method $private:pf$ $lid:id$ : $typ:<:ctyp< >>$ = $exp:expr$ >>

   <:class_str_item< method $lid:id$ : $typ:poly_type$ = $exp:expr$ >>
   public method
   <:class_str_item< method $private:Ast.PrNil$ $lid:id$ : $typ:poly_type$ = $exp:expr$ >>

   <:class_str_item< method $private:pf$ $lid:id$ :
   $typ:poly_type$ $pat:pattern$ = $exp:expr$ >>
   method arguments
   <:class_str_item< method $private:pf$ $lid:id$ :
```

```
$typ:poly_type$ = fun $pat:pattern$ -> $exp:expr$ >>

<:class_str_item< method $private:pf$ $lid:id$ :
$typ:poly_type$ : $typ:res_type$ = $exp:expr$ >>
return type constraint
<:class_str_item< method $private:pf$ $lid:id$ :
$typ:poly_type$ = ($exp:expr$ : $typ:res_type$) >>

<:class_str_item< method $private:pf$ $lid:id$ :
$typ:poly_type$ :> $typ:res_type$ = $exp:expr$ >>
return type coercion
<:class_str_item< method $private:pf$ $lid:id$ :
$typ:poly_type$ = ($exp:expr$ :> $typ:res_type$ ) >>

<:class_str_item< value $mutable:mu$ $lid:id$ = $exp:expr$ >>
non-overriding instance variable
<:class_str_item< value $!:Ast.OvNil$ $mutable:mf$ $lid:id$ = $exp:expr$ >>

<:class_str_item< value $!:override$ $lid:id$ = $exp:expr$ >>
immutable instance variable
<:class_str_item< value $!:override$ $mutable:Ast.MuNil$ $lid:id$ = $exp:expr$ >>

<:class_str_item< value $!:override$ $mutable:mf$ $lid:id$ :
$typ:res_type$ = $exp:expr$ >>
type restriction<:class_str_item
< value $!:override$ $mutable:mf$ $lid:id$ =
($exp:expr$ : $typ:res_type$) >>

<:class_str_item< value $!:override$ $mutable:mf$ $lid:id$ :>
$typ:res_type$ = $exp:expr$ >>
simple value coercion
<:class_str_item< value $!:override$ $mutable:mf$ $lid:id$ =
($exp:expr$ :> $typ:res_type$) >>

<:class_str_item< value $!:override$ $mutable:mf$ $lid:id$ :
$typ:expr_type$ :> $typ:res_type$ = $exp:expr$ >>
complete value coercion
<:class_str_item< value $!:override$ $mutable:mf$ $lid:id$ =
($exp:expr$ : $typ:expr_type$ :> $typ:res_type$) >>

<:class_str_item< method virtual $lid:id$ : $typ:poly_type$ >>
public virtual method
<:class_str_item< method $private:Ast.PrNil$ virtual $lid:id$ : $typ:poly_type$ >>

<:class_str_item< value $!:override$ virtual $lid:id$ : $typ:type$ >>
immutable virtual value
<:class_str_item< value $!:override$ virtual $mutable:Ast.MuNil$ $lid:id$
```

```
   : $typ:type$ >>

   A missing superclass binding is represented with the empty string
   as identifier.  Normal methods and explicitly polymophically typed
   methods are represented with the same ast node (CrMth). For a normal
   method the poly_type field holds the empty type (TyNil).
*)
];
```

Listing 35: Camlp4 Ast Definition

4.5 TestFile

Some test files are pretty useful(in the distribution of camlp4) like `test/fixtures/macro_test.ml`.

Chapter 5

Libraries

5.17 Modules

Chapter 6

Runtime

Chapter 7

GC

Should be re-written later

Chapter 8

Object-oriented

Write later

Chapter 9

Language Features

write later

9.9 The module Language

Chapter 10

subtle bugs

Chapter 11

Interoperating With C

Write later

Chapter 12

Pearls

Chapter 13

Compiler

Chapter 14

XX

Chapter 14

Topics