Camlp4
version 2.04

Daniel de Rauglaudre

November 27, 1999

Copyright © 1999 Institut National de Recherche en Informatique et Automatique

Contents

1 Camlp4 **5**
- 1.1 Camlp4 predefined files . . . 5
- 1.2 Preprocessing Objective Caml . . . 7
  - 1.2.1 The compiler . . . 7
  - 1.2.2 Remark . . . 7
  - 1.2.3 The toplevel . . . 8
- 1.3 User syntax modifications . . . 8

2 Quotations **9**
- 2.1 Syntax . . . 9
- 2.2 Creating a quotation expander . . . 10
- 2.3 Errors . . . 10
  - 2.3.1 Exception raised in an expander . . . 10
  - 2.3.2 Error while parsing the expander result (string expanders) 10
  - 2.3.3 Typing error . . . 11
- 2.4 Predefined quotation expanders . . . 11
- 2.5 Some examples . . . 11
  - 2.5.1 Defining constants . . . 11
  - 2.5.2 Lambda Calculus . . . 12

3 Grammars **15**
- 3.1 Grammars and entries . . . 15
- 3.2 Extension . . . 15
- 3.3 Language parsed . . . 18
- 3.4 Deletion . . . 18
- 3.5 Examples . . . 18
  - 3.5.1 Arithmetic calculator . . . 18
  - 3.5.2 Camlp4 bootstrap . . . 19

4 Syntax extensions **21**
- 4.1 The abstract syntax tree . . . 21
- 4.2 Extending the language . . . 21
- 4.3 Examples of language extensions . . . 22
  - 4.3.1 Infix . . . 22

4.3.2 Repeat until la Pascal . . . 22
4.4 Redefining the whole syntax . . . 23

5 The Revised syntax 25
5.1 Phrases . . . 25
5.2 Imperative constructions . . . 25
5.3 T-uples and lists . . . 26
5.4 Irrefutable patterns . . . 27
5.5 Constructions with matching . . . 27
5.6 Mutables and assignment . . . 28
5.7 Types . . . 28
5.8 Modules . . . 30
5.9 Objects . . . 30
5.10 Miscellaneous . . . 30
5.11 Streams and parsers . . . 31

6 Camlp4 library modules 33
6.1 Module `MLast`: abstract syntax tree . . . 33
6.2 Module `Grammar`: extensible grammars . . . 33
6.3 Module `Pcaml`: language grammar, entries and printers. . . . 37
6.4 Module `Plexer`: lexical analyzer . . . 38
6.5 Module `Quotation`: quotation operations . . . 39
6.6 Module `Stdpp`: standard definitions . . . 40
6.7 Module `Token`: tokens for grammars . . . 40

A Quotations for creating abstract syntax trees 43
A.1 General rules . . . 43
A.2 Quotation `expr` . . . 43
A.3 Quotation `patt` . . . 44
A.4 Quotation `ctyp` . . . 45
A.5 Quotation `sig_item` . . . 46
A.6 Quotation `str_item` . . . 46
A.7 Quotation `module_type` . . . 47
A.8 Quotation `module_expr` . . . 47

Introduction

Camlp4 is a Pre-Processor-Pretty-Printer for Objective Caml. It offers syntactic tools (parsers, extensible grammars), the ability to extend the concrete syntax of Objective Caml (quotations, syntax extensions), and to redefine it from scratch.

This document holds the directions for use of Camlp4 and describes these features. It is written for programmers who already know Objective Caml, and are interested in concrete syntax.

Here are the contents of the present document:

- Chapter 1. "Camlp4" describes the Camlp4 features, the predefined syntaxes offered and how to use Camlp4 with the Objective Caml compiler and toplevel.
- Chapter 2. "Quotations" describes the Camlp4 quotation system, the first and simplest way to make syntax extensions.
- Chapter 3. "Grammars" describes the Camlp4 grammar system, a way to create grammars, more powerful and convenient than the simple stream parsers, and which are extensible what yacc and lex are not. Camlp4's predefined syntaxes use this grammar system.
- Chapter 4. "Syntax extensions" describes how to make concrete syntax extensions in Objective Caml and how to create a new concrete syntax for the whole language. A knowledge of the Camlp4 grammar system (chapter 3) is necessary to read this chapter.
- Chapter 5. "The revised syntax" describes an alternative concrete syntax for Objective Caml supposed to fix some difficulties of the standard Objective Caml concrete syntax. This chapter can be read directly without having read the other ones.
- Chapter 6. "Camlp4 library modules" describes the interfaces of the Camlp4 library modules used by the Camlp4 tools: "`Grammar`", "`Pcaml`", "`Plexer`", "`Stdpp`", "`Token`".
- Appendix A. "Quotation for creating abstract syntax trees" describes a quotation expander provided to create language syntax trees.

The "4" in "Camlp4" comes from the four "P" in its definition "Pre-Processor-Pretty-Printer".

Chapter 1

Camlp4

Camlp4 is a Pre-Processor-Pretty-Printer for Objective Caml, parsing a source file and printing some result.

To run, Camlp4 needs to load files holding operations defining or modifying parsing (how to read the input file) and printing (how to print the result). Such files are Objective Caml object files (ending with "`.cmo`" or "`.cma`").

The command "`camlp4`" runs the Camlp4 program. The first options are objects ("`.cmo`") or library ("`.cma`") files which are loaded, selecting different parsing and printing operations. Among these first options, it is also possible to use the option "`-I`" to select directories where objects and library files are searched. But, important remark: by default, the system *does not* searches in the current directory: to select some object file, e.g. "`foo.cmo`" in the current directory, write "`./foo.cmo`" or add the option "`-I .`".

1.1 Camlp4 predefined files

The system provides some files, installed in *Camlp4 library directory* (depending on the installation, usually `/usr/local/lib/camlp4`: in the whole document, this directory is named `camlp4-lib-dir`). The provided files are:

- Parsing syntax:
  - `pa_o.cmo`: Objective Caml.
  - `pa_r.cmo`: Revised: this is an alternative syntax for the language (chapter 5).
- Printing syntax:
  - `pr_o.cmo`: Pretty print with Objective Caml syntax. Add an option `-l` to the command line to specify a line length.
  - `pr_r.cmo`: Pretty print with revised syntax. Add this option `-l` too.
  - `pr_dump.cmo`: Dump the syntax tree

Other predefined files allow to extend parsing and printing operations:

- Parsing extensions:
  - `pa_oop.cmo`: (to be loaded after "`pa_o.cmo`") Parsers and streams, with Objective Caml syntax, without code optimization, corresponding to what is internally generated by the Objective Caml compiler.
  - `pa_op.cmo`: (to be loaded after "`pa_o.cmo`") Parsers and streams, with Objective Caml syntax, with code optimization.
  - `pa_rp.cmo`: (to be loaded after "`pa_r.cmo`") Parsers and streams, with revised syntax and code optimization.
  - `pa_extend.cmo`: (to be loaded either after "`pa_o.cmo`" or "`pa_r.cmo`") Syntax extensions for the Camlp4 grammar system (Chapter 3).
  - `pa_ifdef.cmo`: add `ifdef` statement like in C (conditional compilation), in expressions or structure items. Add the options `-D` and `-U` to the camlp4 command line. To define the symbol `FOO`, either use "`-DFOO`" in the camlp4 command line, e.g.:

    ```
    camlp4o pa_ifdef.cmo -DFOO file.ml
    ```

    or make a file, e.g. "`foo.ml`", holding

    ```
    Pa_ifdef.define "FOO";;
    ```

    compile it and load its corresponding object file:

    ```
    camlp4o pa_ifdef.cmo ./foo.cmo file.ml
    ```

    To undefine the symbol `FOO`, use either "`-UFOO`" or the function "`Pa_ifdef.undef`". The statement "`ifdef`" can be used in a place of a struture item or an expression: in this last case, "`else`" is required:

    ```
    ifdef FOO then type t = A of int;;
    let f x = ifdef FOO then x + 1 else x - 1;;
    ```
  - `pa_ru.cmo`: (to be loaded after "`pa_r.cmo`" only) Add type constraints of type `unit` inside sequences.
- Printing extensions:
  - `pr_op.cmo`: Try to rebuilt parsers and streams in Objective Caml syntax.
  - `pr_rp.cmo`: Try to rebuilt parsers and streams in revised syntax.
  - `pr_extend.cmo`: Try to rebuilt EXTEND instructions.
  - `pr_depend.cmo`: Print file dependencies.

Some shortcut commands are provided:

- "`camlp4o`" is equivalent to:

```
camlp4 pa_o.cmo pa_op.cmo pr_dump.cmo
```

- "`camlp4r`" is equivalent to:

```
camlp4 pa_r.cmo pa_rp.cmo pr_dump.cmo
```

1.2 Preprocessing Objective Caml

1.2.1 The compiler

The standard use of Camlp4 is as a preprocessor for Objective Caml with the printer "`pr_dump.cmo`". The Objective Caml compiler "`ocamlc`" recognizes it is a dump, and does not apply another parsing. In case of typing errors, the error is normally showed in the input file.

We remind that the command "`ocaml`" has an option "`-pp`" for calling a preprocessor which automatically deals with intermediate files (see the documentation of Objective Caml). So, the normal use of the combinaison Camlp4-Objective Caml is done by the command:

```
ocamlc -pp "camlp4 camlp4-options" ocaml-options input-files
```

It is often necessary to add the *Camlp4 library directory* in ocaml's path, using option "`-I`". For information, the Camlp4 command has an option "`-where`" echoing the full name of this directory.

1.2.2 Remark

It is possible to use Camlp4 with "`pr_o.cmo`", instead of "`pr_dump.cmo`" for Objective Caml preprocessing, but it is not recommended, for:

- The parsing would be done twice: one of the user input file by Camlp4, which pretty prints an intermediate file (which takes some time), and one of the result of Camlp4 by the Objective Caml parser.
- Objective Caml may find more syntax errors, for Camlp4 with `pa_o.cmo` accepts a syntax larger than Objective Caml's (some syntax restriction is done in `pr_dump.cmo`)
- In case of typing errors, Objective Caml would show the error location in the resulting file of Camlp4, what is not convenient.

However, there is a case where it is interesting to use "`pr_o.cmo`" for objective Caml preprocessing. When the error is located in a quotation (chapter 2) or in a syntax extension (chapter 4), it is not always obvious to understand where the error is exactly located, and there may be no location for the error. Using temporarily camlp4 with "`pr_o.cmo`" allows to see the exact location of the error, which can be understood by the programmer of the quotation expander or the syntax extension.

1.2.3 The toplevel

The Objective Caml toplevel, "`ocaml`" can use Camlp4. To do this, it must be started with the option "`-I` *Camlp4 library directory*", or, under Unix, "`-I ‘camlp4 -where‘`". Then, one must load:

- Either the file "`camlp4o.cma`" to continue with Objective Caml syntax.
- Or the file "`camlp4r.cma`" to continue with the Revised syntax.

To change the behavior of Camlp4, just load the needed module with the directives "`#load`" and "`#use`": for example, each load of "`pa_o.cmo`" or "`pa_r.cmo`" would change the current input syntax to be used in the next sentences.

The files included by the directive "`#use`" are treated by the current Camlp4 syntax too. Important remark: a syntax modification takes place after the complete load of a file, not after each sentence inside the file.

It is not possible to load a Camlp4 printing syntax file in the toplevel. The printing results are done by the Objective Caml toplevel in standard Objective Caml syntax.

1.3 User syntax modifications

The next chapters explain the different ways to create files holding syntax modifications. The simplest one is the quotations system (chapter 2).

To introduce other syntax extensions of the language, or to redefine the complete language syntax, one must use the Camlp4 Grammar System (chapter 3). The following chapter shows how to extend or redefine the language syntax (chapter 4).

It is not possible for the user to extend the Camlp4 printing: its main usage is through the file "`pr_dump.cmo`", the other printing files, displaying source text, are just provided for convenience and their behaviors are not (yet) modifyable.

Chapter 2

Quotations

This chapter presents the Camlp4 quotation system. A quotation is a part of a program enclosed by special parentheses in position of expression or pattern, and whose treatment is done by an user function, called a "quotation expander". Several quotations expanders are usable in the same source code.

When Camlp4 encounters such a construction, the appropriate quotation expander is called with the string contents of the quotation.

There are two kinds of quotation expanders:

- Those returning *strings*. The returned string is parsed by Camlp4 afterwards. They are easy to use, but they depend on the language syntax: the returned string must be of Ocaml syntax if the language syntax used is Ocaml's, in Revised syntax if the language syntax is the Revised one, etc.

- Those returning *syntax trees*. They are independant from the language syntax used, but to use them, one muse know how to create syntax tree nodes. See appendix A.

2.1 Syntax

There are two kinds of quotations:

- The anonymous quotations start with "`<<`" and end with "`>>`". The associated expander is the "default" expander.

- The named quotations start with "`<:id<`" and end with "`>>`", where "`id`" is any identifier. The associated expander is the expander named "`id`".

The string inside a quotation is any string. It can hold other quotations. The string sequences "`<<`", "`<:id<`" and "`>>`", if not matching, must be prefixed by a backslash and backslashes must be doubled.

2.2 Creating a quotation expander

A quotation expander is a function written in Objective Caml. A call to the Camlp4 library function "`Quotation.add`" (see section 6.5) adds the quotation expander. The variable "`Quotation.default`" holds the name of the default quotation.

The file holding the quotation expander must be compiled using the Objective Caml compiler, with the *Camlp4 library directory* in its directory path (option "`-I`"). An object file (ending with "`.cmo`") is created, which is loadable in Camlp4 or in the Objective Caml toplevel using the directive "`#load`". The source can be loaded directly using the directive "`#use`".

2.3 Errors

If some kind of error happens while compiling a quotation, Camlp4 or Objective Caml display an error message, which shows the source position of the error. By default, the error message highlights the whole quotation, but a quotation expander can specify more precise "locations" while parsing and generating its returning string.

A "location" is a couple of integers: the first one is the position of the first character of the concerned text, the second one the position of the first character after the concerned text.

2.3.1 Exception raised in an expander

To raise an exception in an expander, it is possible to use the function "`Stdpp.raise_with_loc`" instead of "`raise`", this first function having a supplementary parameter which is the location of the error relative to the begin of the quotation.

2.3.2 Error while parsing the expander result (string expanders)

A problem can arise in *string* expanders (those returning *strings*). The resulting string may hold syntax errors. In this case, for there is no "input location" for the error, and it is not directly possible to see this error.

To solve this problem, a special option of Camlp4 can be used: "`-QD`" (see the manual page). It is also possible to set the variable "`Pcaml.quotation_dump_file`" (see section 6.3). The offending string is then written in a file (whose name is specified by this option or this variable) and the error message gives the location of the error in this file.

This problem cannot arise in *syntax trees* expanders, for there is no parsing following the expansion.

2.3.3 Typing error

It is possible to specify precise source locations to some parts (antiquotations) of the expander resulting string. For this, there are to ways depending on the expander kind:

- In a *string* expander, enclose the resulting string with a location directive, which will be interpreted by the language lexer. Its syntax recognized by the default language lexer (`Plexer`), is:

  location_directive `::= $` *int* `:` *string* `$`

  The *int* value is the source location of the first character of the antiquotation, relative to the beginning of the quotation string. The *string* is the antiquotation itself.

- In a *syntax tree* expander, the locations of the antiquotations must be relative to the beginning of the antiquotation, and the tree generated must be enclosed with a *location node* (see appendix sections A.2 and A.3). Remark: the subtrees not enclosed with this location node are relocated with the entire quotation after expansion.

2.4 Predefined quotation expanders

A quotation expander is provided in the file "`q_MLast.cmo`" allowing to create abstract syntax tree nodes from concrete revised syntax. It is used in language syntax extensions (chapter 4).

Another quotation expander provided is "`q_phony.cmo`". It transforms all quotations into variables whose name holds all the text of the quotation, `<<` and `>>` included. It is useful for pretty printing programs with the quotations not expanded.

2.5 Some examples

2.5.1 Defining constants

This is a way to represent constants by some name. The quotation expander is just a string matching. Here is the file "`foo.ml`":

```
let exp _ =
  function
    "keyPressEvent" -> "2"
  | "exposeEvent" -> "12"
  | "configureNotifyEvent" -> "22"
;;
Quotation.add "const" (Quotation.ExStr exp);;
Quotation.default := "const";;
```

The compilation command is:

```
ocamlc -c -I `camlp4 -where` foo.ml
```

Now, the following piece of program, in some file "bar.ml"

```
match event.typ with
  <<configureNotifyEvent>> -> ...
| <<exposeEvent>> -> ...
| <<keyPressEvent>> -> ...
```

will be interpreted like:

```
match event.typ with
  22 -> ...
| 12 -> ...
| 2 -> ...
```

It can be compiled using the command:

```
ocamlc -pp "camlp4o ./foo.cmo" bar.ml
```

or pretty printed:

```
camlp4o pr_o.cmo ./foo.cmo bar.ml
```

2.5.2 Lambda Calculus

This is another example, more complicated, for treating "lambda calculus terms". A type "term" is defined as:

```
type term =
    Abs of term
  | Ref of int
  | App of term * term
;;
```

but instead of manipulating terms, like:

```
Abs (Ref 0)
Abs (Abs (Ref 1))
App (Abs (App (Ref 0, Ref 0)), Abs (App (Ref 0, Ref 0)))
Abs (App (Abs (Ref 0), Abs (App (Ref 1, Ref 0))))
```

we want to represent them by a concrete syntax in quotations:

```
<< [x]x >>
<< [x,y]x >>
<< ([x](x x) [x](x x)) >>
<< [x]([y]y [z](x z)) >>
```

The details of the complete syntax of lambda terms is not described here. What is important is that the users of lambda terms prefer to write in their programs this concrete syntax instead of a combination of values of type term, harder to understand.

Here, we choose to implement the quotation expander using a Camlp4 grammar (see chapter 3). Just have a fast look of this code for the moment and return here when you have read the chapter on grammars:

```
type lambda =
    Lam of string * lambda
  | App of lambda * lambda
  | Var of string
  | Antiquot of string
;;

let gram = Grammar.create (Plexer.make ());;
let lambda_eoi = Grammar.Entry.create gram "lambda";;
let lambda = Grammar.Entry.create gram "lambda";;
EXTEND
   lambda_eoi: [[ x = lambda; EOI -> x ]];
   lambda: [[ "["; x = LIDENT; "]"; t = lambda        -> Lam (x, t)
            | "("; t1 = lambda; t2 = lambda; ")"      -> App (t1, t2)
            | x = LIDENT                              -> Var x
            | "^"; x = LIDENT                         -> Antiquot x
            | "["; x = LIDENT; ","; l = LIST1 LIDENT SEP ","; "]";
              t = lambda
                -> List.fold_right (fun x t -> Lam (x, t)) (x::l) t ]];
END;;

let rec index x =
  function
    [] -> raise Not_found
  | y::l -> if x = y then 0 else succ (index x l)
;;

let rec convert env =
  function
    Lam x t ->
      "Abs (" ^ convert (x::env) t ^ ")"
  | App t1 t2 ->
      "App (" ^ convert env t1 ^ ", " ^ convert env t2 ^ ")"
  | Var x ->
      let i =
        try index x env with
          Not_found -> failwith ("unbound variable " ^ x)
      in
```

```
        "Ref " ^ string_of_int i
    | Antiquot x -> x
;;

let lam_exp _ str =
  let cs = Stream.of_string str in
  let t = Grammar.Entry.parse lambda_eoi cs in
  convert [] t
;;

Quotation.add "term" (Quotation.ExStr lam_exp);;
Quotation.default := "term";;
```

This source code, holding an Objective Caml syntax extension (keyword "`EXTEND`") for grammars, must be preprocessed by Camlp4:

```
ocamlc -pp "camlp4o pa_extend.cmo" -c -I `camlp4 -where` foo.ml
```

and files holding term quotations can be compiled using the command:

```
ocamlc -pp "camlp4o ./foo.cmo" -c -I `camlp4 -where` bar.ml
```

Chapter 3

Grammars

Camlp4's grammar system is a way to parse streams with more power and more convenience than simple stream parsers. A grammar deals with operator precedences and associativity, and left factorization, and its entries are extensible.

3.1 Grammars and entries

The Camlp4 library provides a module "`Grammar`" and a syntax extension file "`pa_extend.cmo`" to program grammars. All details on modules, types and functions are described chapter 6.

A grammar can be created using the function "`Grammar.create`". It takes a lexer as parameter. A good candidate it the function "`Plexer.make`" but the user can creates its own lexer, providing that it is of type "`Token.lexer`".

A grammar holds entries. An entry can be created using the function "`Grammar.Entry.create`". The first parameter is the grammar, the second one is a string used to name the entry in error messages and in entries dump. Entries are created empty, i.e. raising "`Stream.Error`" when called. An entry is composed of entry precedence levels, the first one being the least precedent and the last one the most.

Lastly, a stream can be parsed by en entry using the function "`Grammar.Entry.parse`". In case of syntax error, the exception "`Stream.Error`" is raised, encapsulated by the exception "`Stdpp.Exc_located`", giving the location of the error.

3.2 Extension

An entry can be extended using the function "`Grammar.extend`". But its interface being quite complicated and, as it must be used with appropriate type constraints, the Camlp4 library provides a file, named "`pa_extend.cmo`", compatible with "`pa_o.cmo`" and "`pa_r.cmo`" which creates a new instruction doing this work.

This instruction is "EXTEND" which has the following format:

```
EXTEND
   { GLOBAL : global-list ; }
   entry : { position } extension ;
   ...
   entry : { position } extension ;
END
```

EXTEND, GLOBAL and END are keywords. There are some other keywords in this instruction, all in uppercase.

The *entry* is a name of an entry variable (simple identifier or prefixed by a module name). It is extended according to the rules given by *extension*. The optional *position* tells how to insert the extension among the entry precedence levels: by default, the extension is done at the first precedence level (creating this level if the entry is empty).

The optional *global-list* restricts the list of the *global* entries (separated by spaces) extended in the "EXTEND" instruction. The other entries are created locally. By default, all entries are global and must correspond to entry variables visible at the "EXTEND" instruction point.

Syntax of a position

A *position* can be:

- FIRST: The *extension* is inserted at the beginning of the precedence levels.
- LAST: The *extension* is inserted as the end of the precedence levels.
- BEFORE *label*: The *extension* is inserted before the precedence level so labelled.
- AFTER *label*: The *extension* is inserted after the precedence level so labelled.
- LEVEL *label*: The *extension* is inserted at the precedence level so labelled.

Only the case LEVEL extends already existing levels: the other cases create new levels.

Syntax of an extension

The syntax of an entry *extension* is:

```
[   { label } { associativity } level-rules
|   ...
|   { label } { associativity } level-rules   ]
```

The optional *label* identifies the current level for possible future extensions. The optional *associativity* can be LEFTA, RIGHTA or NONA for respectively left, right and no associativity: the default is left associativity.

Syntax of a level-rules

The *level-rules* have the format:

```
[   { pattern = } symbol ; ... { pattern = } symbol { -> action }
|   ...
|   { pattern = } symbol ; ... { pattern = } symbol { -> action }   ]
```

The *pattern* is a language simple pattern, limited to identifiers, wildcard (character "underscore") and tuples of patterns. If not present, it is equivalent to the wildcard.

The *action* is a language expression where the patterns are bound to the result of their corresponding *symbol*; in addition, the variable "`loc`" is bound to the source location of the rule. The action part is optional; by default it is the value "`()`".

A *symbol* is either:

A string meaning: a keyword. If this keyword does not exist in the current grammar, it is added. The syntax of the keyword must respect the rules of the lexer. Its type is `string`.

An entry name (simple identifier or prefixed by a module name) meaning a call to this entry. Its type is the entry result type.

The keyword SELF meaning the entry being extended (first level).

The keyword NEXT meaning the entry being extended at the next level.

A token pattern. Tokens patterns are identifiers starting with an uppercase letter optionally followed by a string; they recognize tokens generated by the lexer. See for example, the constructors generated by the predefined lexer (section 6.4), lexing Ocaml normal and revised syntaxes.

A metasymbol among:

- `LIST0` and `LIST1` whose syntax is:

  `LISTx` *symbol1* { `SEP` *symbol2* }

  meaning a list (potentially empty for `LIST0` and with at least one element for `LIST1`) of *symbol1*, whose elements are optionally separated by *symbol2*. The type is `t1 list` where `t1` is the type of *symbol1* (the result of the optional *symbol2* is lost).

- `OPT` followed by a *symbol*, meaning this symbol or nothing. The type is `t option` where `t` is the type of *symbol*.

A level-rules meaning an anonymous entry with only one level.

A symbol between parentheses

3.3 Language parsed

Entries are stream parsers improved. Each precedence level can be seen as an independent extensible stream parser.

In a precedence level, the rules are factorized, i.e. several rules can start with the same symbols, and the parser is represented by a tree of symbols where leaves are either semantic actions, or dead ends.

When an extension is done, the new rules are inserted in the precedence level tree. The system does not check whether the entry precedence level will "work", nor whether it would work together with the other levels and the other entries.

This insertion is generally done at the end of the rules list. The special cases are for symbols of type token, which are always inserted before the other symbols (non terminals, lists, options, etc), tokens with parameters before tokens without parameters (parameter is the empty string).

There is no factorization of rules between different levels nor, a fortiori, with other entries.

While parsing an entry, the system tries the first precedence level. Then, if it fails, it tries the second one, etc. If the last one fails, the entry fails and the caller of this entry resumes, either by raising a parse error (the parsing is then stopped), or by trying other rules.

3.4 Deletion

Entries rules can be deleted using the function "`Grammar.delete_rule`". But, like for "`Grammar.extend`", it is not documented. One must use the instruction "`DELETE_RULE`", generating a call to this function. This instruction is a syntax extension, loaded together with the instruction "`EXTEND`" by the file "`pa_extend.cmo`".

The syntax of "`DELETE_RULE`" is:

```
DELETE_RULE
   entry : symbol ; ... symbol
END
```

The rule defined by the list of symbols (semantic actions forgotten), if found, is deleted.

3.5 Examples

3.5.1 Arithmetic calculator

This is an example of a grammar of arithmetic expressions:

```
let gram = Grammar.create (Plexer.make ());;
let test = Grammar.Entry.create gram "expression";;
let expr = Grammar.Entry.create gram "expression";;
EXTEND
  test: [ [ e = expr; EOI -> e ] ];
  expr: [ "plus" LEFTA
          [ e1 = expr; "+"; e2 = expr -> e1 + e2
          | e1 = expr; "-"; e2 = expr -> e1 - e2 ]
        | "mult" LEFTA
          [ e1 = expr; "*"; e2 = expr -> e1 * e2
          | e1 = expr; "/"; e2 = expr -> e1 / e2 ]
        | [ e = INT -> int_of_string e
          | "("; e = expr; ")" -> e ] ];
END;;
let calc str =
  try Grammar.Entry.parse test (Stream.of_string str) with
    Stdpp.Exc_located (loc, e) ->
      Printf.printf "Located at (%d, %d)\n" (fst loc) (snd loc);
      raise e
;;
```

Now, an extension of the entry "`expr`" to add the modulo, could be:

```
EXTEND expr:
  AFTER "mult" [ [ e1 = expr; "mod"; e2 = expr -> e1 mod e2 ] ];
END;;
```

3.5.2 Camlp4 bootstrap

The Camlp4 parsers of Objective Caml and revised syntax are implemented using the present grammar system: their sources files "`pa_o.ml`" and "`pa_r.ml`", in Camlp4's distribution, hold `EXTEND` instructions defining the whole language parsers. This make them extensible. See chapter 4.

Moreover, the syntax extension adding the instruction "`EXTEND`" is programmed in the file "`pa_extend.ml`" in Camlp4's distribution, as a syntax extension of the language parser. It contains code looking like:

```
EXTEND
  expr: [ [ "EXTEND"; g = OPT global; el = LIST0 entry; "END" -> ...
  global: [ [ "GLOBAL"; ....
  entry: [ [ e = LIDENT; ":"; p = OPT position; ....
  ...
END
```

So, the Camlp4 sources use widely the Camlp4 facilities. The very first compilation of Camlp4 is done by pure Objective Caml versions of Camlp4's sources created by Camlp4.

Chapter 4

Syntax extensions

This chapter introduces the way to extend the syntax of the language, and how to redefine the whole syntax of the language.

4.1 The abstract syntax tree

A syntax extension or redefinition holds grammar rules which must return abstract syntax tree nodes. The provided file "`q_MLast.cmo`" holds quotations expanders allowing to create such nodes using a concrete syntax: the description of the abstract syntax tree is not necessary. The complete description of these quotations is given in appendix A.

4.2 Extending the language

The language predefined syntaxes, "`pa_o.cmo`" and "`pa_r.cmo`" use the Camlp4 grammar system: so they are extensible.

The module "`Pcaml`" holds grammars and entries variables used by the langage predefined syntaxes files. Syntax extensions can be done by extensions of the appropriate entries.

In some entries, "`pa_o.cmo`" and "`pa_r.cmo`" define labels allowing to locate some levels and to add new rules relative to these levels. The entries and their labels are:

- "`Pcaml.expr`", for expressions, with the labels:
  - "`let`" corresponding to the level of the instruction "`let...in`".
  - "`||`", "`&&`", "`<`", "`+`", "`*`", "`**`" corresponding to the so named operator.
  - "`apply`" corresponding to the curryfied application.
  - "`simple`" corresponding to simple expressions.

- "`Pcaml.patt`", for patterns, with the label "`simple`" corresponding to simple patterns.
- "`Pcaml.ctyp`", for types, with the label "`simple`" corresponding to simple types.
- "`Pcaml.sig_item`" and "`Pcaml.str_item`", for signature and structure items, with only one level labelled "`top`".
- "`Pcaml.module_type`" and "`Pcaml.module_expr`", for module types and expressions.
- Some other labels exist but are not documented.

4.3 Examples of language extensions

All examples must be compiled using "`camlp4`". As they use the "`EXTEND`" instruction, Camlp4 must load "`pa_extend.cmo`". As they use the quotations to create abstract syntax tree nodes, Camlp4 must load "`q_MLast.cmo`". The compilation command is therefore:

```
ocamlc -pp "camlp4o pa_extend.cmo q_MLast.cmo" -I camlp4-lib-dir -c file.ml
```

After this, the syntax extension takes place in Camlp4 when "`file.cmo`" is loaded.

4.3.1 Infix

This is an example to add the infix operator "`o`", composition of two functions. For the meaning of the quotation `expr` used here, see appendix A.

```
open Pcaml;;
EXTEND
  expr: AFTER "apply"
    [[ f = expr; "o"; g = expr -> <:expr< fun x -> $f$ ($g$ x) >> ]];
END;;
```

4.3.2 Repeat until la Pascal

The "repeat...until" loop of Pascal is closed to the "while" loop except that it is executed at least once. We can implement it like this:

```
open Pcaml;;
EXTEND
  expr: LEVEL "let"
    [[ "repeat"; e1 = expr; "until"; e2 = expr ->
          <:expr< do $e1$; return while not $e2$ do $e1$; done >> ]];
END;;
```

4.4 Redefining the whole syntax

To redefine the whole syntax, one must start with an empty grammar and empty entries and extend them. The library module "`Grammar.Unsafe`" hold functions to clean up the grammars and the entries.

Examples can be found in the distribution: the files "`meta/pa_r.ml`" and "`etc/pa_o.ml`" which are the sources of "`pa_r.cmo`" and "`pa_o.cmo`".

Chapter 5

The Revised syntax

The revised syntax is an alternative syntax for Objective Caml. Its purpose is to be simpler, more regular, more logical than the normal syntax, and fix some problems which sometimes result in bugs not always easy to find. This syntax was named "righteous" instead of "revised" in previous versions.

This chapter presents the changes from Objective Caml to this syntax.

5.1 Phrases

- A simple semicolon ends the sentences, in structures and in signatures. The double semicolon is not a token. There is no ambiguity with the sequence, which has a special construction (see 5.2).

- The declaration of a global variable is introduced by the keyword "`value`", "`let`" being reserved to the construction "`let..in`":

Ocaml	Revised
`let x = 23;;`	`value x = 23;`
`let x = 23 in x + 7;;`	`let x = 23 in x + 7;`

- In interfaces, one must use "`value`", too, instead of "`val`".

Ocaml	Revised
`val x : int;;`	`value x : int;`

5.2 Imperative constructions

- The sequence is introduced by the keyword "`do`". Each instruction of the sequence, except the last one, ends with a semicolon. The sequence is terminated by the keyword "`return`" followed by the last intruction:

Ocaml	Revised
`e1; e2; e3; e4`	`do e1; e2; e3; return e4`

- In the body of "`for`" and "`while`" loops, all instructions end with a semicolon, the last one included:

Ocaml	Revised

`while e1 do`
`  e2; e3; e4`
`done`
	`while e1 do`
	`  e2; e3; e4;`
	`done`

- It is possible to write an empty sequence, equivalent to the last expression alone, a "`while`" loop and a "`for`" loop with an empty body, equivalent to a loop where the body is "`()`":

```
do return e
while e do done
for i = e1 to e2 do done
```

5.3 T-uples and lists

- Parentheses are required in t-uples:

Ocaml	Revised
`1, "hello", World`	`(1, "hello", World)`

- Lists are always enclosed with "`[`" and "`]`". Their syntax is:

list ::= `[` *elem-list opt-cons* `]`
elem-list ::= *expression* `;` *elem-list* | *expression*
opt-cons ::= `::` *expression* | *(*empty*)*

A list is a sequence of expressions separated by semicolons, optionally ended by a "`::`" and an expression, the whole being enclosed by brackets. Examples:

Ocaml	Revised
`x::y`	`[x::y]`
`[x; y; z]`	`[x; y; z]`
`x::y::z::t`	`[x::[y::[z::t]]]`
`x::y::z::t`	`[x; y; z :: t]`

Note the two ways to write the last case.

5.4 Irrefutable patterns

There is a notion of "irrefutable patterns" used by some syntactic constructions. Matching against these patterns never fails. An "irrefutable pattern" is either:

- A variable.
- The wildcard "`_`".
- The constructor "`()`".
- A t-uple with irrefutable patterns.
- A record with irrefutable patterns.
- An irrefutable pattern with a type constraint.

Note that the term "irrefutable" does not apply to all patterns which never fail: constructors alone in their type declarations, except "`()`", are not said "irrefutable".

5.5 Constructions with matching

- The keyword "`function`" no longer exists. One must use only "`fun`".
- The pattern matchings, in constructions with "`fun`", "`match`" and "`try`" are closed with brackets: an open bracket "`[`" before the first case, and a close bracket "`]`" after the last one:

Ocaml	Revised

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

`fun x -> x;;`	`fun [x -> x];`

But if there is only one case and if the pattern is *irrefutable*, the brackets are not required. These examples work identically in Ocaml and in revised syntax:

Ocaml	Revised
`fun x -> x` `fun {foo=(y, _)} -> y`	`fun x -> x` `fun {foo=(y, _)} -> y`

The currified pattern matching can be done with "`fun`", but only with *irrefutable* patterns:

Ocaml	Revised
`fun x (y, z) -> t` `fun x y (C z) -> t`	`fun x (y, z) -> t` `fun x y -> fun [C z -> t]`

- It is possible to write the empty function, raising the exception "`Match_failure`" whichever parameter is applied, the empty "match", raising "`Match_failure`" after having evaluated its expression, and the empty "try", equivalent to its expression without `try`:

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

- The patterns after "`let`" and "`value`" must be irrefutable. The following Ocaml expression:

```
let f (x::y) = ...
```

  must be written in Revised:

```
let f = fun [ [x::y] -> ...
```

- The construction "`where`" is back, but one can write only one bind: `e where x = y` but not: `e where x = y and z = t`

5.6 Mutables and assignment

- The instruction "`<-`" is written "`:=`":

Ocaml	Revised
`x.f <- y`	`x.f := y`

- The "`ref`" type is declared as a record type with one field named "`val`", instead of "`contents`". The operator "`!`" does not exist any more, and references are assigned like the other mutables:

Ocaml	Revised
`x := !x + y`	`x.val := x.val + y`

5.7 Types

- The type constructors are before their type parameters, which are curried:

Ocaml	Revised

`int list`
`('a, bool) Hashtbl.t`
`type 'a foo =`
`  'a list list;;`
	`list int`
	`Hashtbl.t 'a bool`
	`type foo 'a =`
	`  list (list 'a);`

- The abstract types are represented by a unbound type variable:

Ocaml	Revised
`type 'a foo;;` `type bar;;`	`type foo 'a = 'b;` `type bar = 'a;`

- Parentheses are required in t-uples of types:

Ocaml	Revised
`int * bool`	`(int * bool)`

- In declaration of a concrete type, brackets must enclose the constructor declarations:

Ocaml	Revised
`type t = A of i | B;;`	`type t = [ A of i | B ];`

- It is possible to make the empty type, without constructor:

```
type foo = [];
```

- There is a syntax difference between data constructors with several parameters and data constructors with one parameter of type t-uple.

  The declaration of a data constructor with several parameters is done by separating the types with "`and`". In expressions and patterns, this constructor parameters must be currified:

Ocaml	Revised
`type t = C of t1 * t2;;` `C (x, y);;`	`type t = [ C of t1 and t2 ];` `C x y;`

  The declaration of a data constructor with one parameter of type t-uple is done by using a t-uple type. In expressions and patterns, the parameter has not to be currified, since it is alone:

Ocaml	Revised
`type t = D of (t1 * t2);;` `D (x, y);;`	`type t = [ D of (t1 * t2) ];` `D (x, y);`

- The predefined constructors "`True`" and "`False`" start with an uppercase letter.

- In record types, the keyword "`mutable`" must appear after the colon:

Ocaml	Revised
`type t = {mutable x : t1};;`	`type t = {x : mutable t1};`

5.8 Modules

Modules application uses currification:

Ocaml	Revised
`type t = Set.Make(M).t;;`	`type t = (Set.Make M).t;`

5.9 Objects

The objects also have a revised syntax. To see it, the simplest way is to write examples in normal syntax and to convert them into revised syntax using the command:

```
camlp4o pr_r.cmo file.ml
```

5.10 Miscellaneous

- The "`else`" is required in the "`if`" instruction:

Ocaml	Revised
`if a then b`	`if a then b else ()`

- The boolean operations "or" and "and" must be only written with "`||`" and "`&&`":

Ocaml	Revised
`a or b & c`	`a || b && c`
`a || b && c`	`a || b && c`

- No more "`begin end`" construction. One must use parentheses.
- The operators as functions are written with an backslash:

Ocaml	Revised
`(+)`	`\+`
`(mod)`	`\mod`

- The operators with special characters are not automatically infix. To define infixes, use the syntax extensions (chapter 4).
- It is possible to group together several declarations either in an interface or in an implementation by enclosing them between "`declare`" and "`end`". Example in an interface:

```
declare
  type foo = [ Foo of int | Bar ];
  value f : foo -> int;
end;
```

This can be useful when extending the language structure or signature items with a construction generating several items.

5.11 Streams and parsers

- The streams and the stream patterns are bracketed with "`[:`" and "`:]`" instead of "`[<`" and "`>]`".
- The stream component "terminal" is written with a backquote instead of a quote:

Ocaml	Revised
`[< '1; '2; s; '3 >]`	`[: `1; `2; s; `3 :]`

- The cases of parsers are bracketed with "`[`" and "`]`", like for "`fun`", "`match`" and "`try`". If there is one case, the brackets are not mandatory:

Ocaml	Revised
`parser` `  [< 'Foo >] -> e` `| [< p = f >] -> f`	`parser` `[ [: `Foo :] -> e` `| [: p = f :] -> f]`
`parser [< 'x >] -> x`	`parser [ [: `x :] -> x]`
`parser [< 'x >] -> x`	`parser [: `x :] -> x`

- It is possible to write the empty parser raising the exception "`Stream.Failure`" whichever parameter is applied, and the empty stream matching always raising "`Stream.Failure`":

```
parser []
match e with parser []
```

Chapter 6

Camlp4 library modules

6.1 Module `MLast`: abstract syntax tree

This library module is accessible, but not documented. The creation of abstract syntax tree nodes must be done using quotations, defined in the file `q_MLast.cmo`. See appendix A.

However, some functions are provided to access the "location" field in nodes. They are:

- `loc_of_expr` for expressions.
- `loc_of_patt` for patterns.
- `loc_of_ctyp` for types.
- `loc_of_module_type` for module types.
- `loc_of_module_expr` for module expressions.
- `loc_of_sig_item` for signature items.
- `loc_of_str_item` for structure items.
- `loc_of_phrase` for phrases.

6.2 Module `Grammar`: extensible grammars

```
type g;;
```

The type for grammars, holding entries.

```
val create : Token.lexer -> g;;
```

Create a new grammar, without keywords, using the lexer given as parameter.

```
val tokens : g -> string -> (string * int) list;;
```

Given a grammar and a token pattern constructor, returns the list of the corresponding values currently used in all entries of this grammar. The integer is the number of times this pattern value is used.

Examples:

If the associated lexer uses ("", xxx) to represent a keyword (what is represented by then simple string xxx in an `EXTEND` statement rule), the call `Grammar.token g ""` returns the keywords list.

The call `Grammar.token g "IDENT"` returns the list of all usages of the pattern "IDENT" in the `EXTEND` statements.

```
module Entry :
  sig
    type 'a e;;
    val create : g -> string -> 'a e;;
    val parse : 'a e -> char Stream.t -> 'a;;
    val parse_token : 'a e -> Token.t Stream.t -> 'a;;
    val name : 'a e -> string;;
    val of_parser : g -> string -> (Token.t Stream.t -> 'a) -> 'a e;;
    val print : 'a e -> unit;;
    val find : 'a e -> string -> Obj.t e;;
    external obj : 'a e -> Gramext.g_entry = "%identity";;
  end
;;
```

Module to handle entries.

* `Entry.e` is the type for entries returning values of type `'a`.

* `Entry.create g n` creates a new entry named `n` in the grammar `g`.

* `Entry.parse e` returns the stream parser of the entry `e`.

* `Entry.parse_token e` returns the token parser of the entry `e`.

* `Entry.name e` returns the name of the entry `e`.

* `Entry.of_parser g n p` makes an entry from a token stream parser.

* `Entry.print e` displays the entry `e` using `Format`.

* `Entry.find e s` finds the entry named `s` in `e`'s rules.

* `Entry.obj e` converts an entry into a `Gramext.g_entry` allowing to see what it holds (`Gramext` is visible, but not documented).

```
val of_entry : 'a Entry.e -> g;;
```

Return the grammar associated with an entry.

```
val warning_verbose : bool ref;;
```

Flag for displaying warnings while extension; default = `True`

```
val strict_parsing : bool ref;;
```

Flag to apply strict parsing, without trying to recover errors; default = `False`

Clearing grammars and entries

```
module Unsafe :
  sig
    val reinit_gram : g -> Token.lexer -> unit;;
    val clear_entry : 'a Entry.e -> unit;;
  end
;;
```

Module for clearing grammars and entries. To be manipulated with care, because: 1) reinitializing a grammar destroys all tokens and there may have problems with the associated lexer if it has a notion of keywords; 2) clearing an entry does not destroy the tokens used only by itself.

* `Unsafe.reinit_gram g lex` removes the tokens of the grammar and sets `lex` as a new lexer for `g`. Warning: the lexer itself is not reinitialized.

* `Unsafe.clear_entry e` removes all rules of the entry `e`.

Functions called by statements EXTEND and DELETE RULE

These functions are not to be used directly. The statements `EXTEND` and `DELETE_RULE` use these functions generating type constraints which ensure the typing consistency.

```
val extend :
  (Gramext.g_entry * Gramext.position option *
     (string option * Gramext.g_assoc option *
        (Gramext.g_symbol list * Gramext.g_action) list)
       list)
    list ->
    unit;;
val delete_rule : 'a Entry.e -> Gramext.g_symbol list -> unit;;
```

Functorial interface

Alternative for grammars use. Grammars are not yet Ocaml values: there is no type for them. Modules generated preserve the rule "an entry cannot call an entry of another grammar" by normal Ocaml typing.

```
module type LexerType = sig val lexer : Token.lexer;; end;;
```

The input signature for the functor `Grammar.Make`. `lexer` is the lexer used.

```
module type S =
  sig
    type parsable;;
    val parsable : char Stream.t -> parsable;;
    val tokens : string -> (string * int) list;;
    module Entry :
      sig
        type 'a e;;
        val create : string -> 'a e;;
        val parse : 'a e -> parsable -> 'a;;
        val parse_token : 'a e -> Token.t Stream.t -> 'a;;
        val name : 'a e -> string;;
        val of_parser : string -> (Token.t Stream.t -> 'a) -> 'a e;;
        val print : 'a e -> unit;;
        external obj : 'a e -> Gramext.g_entry = "%identity";;
      end
    ;;
    module Unsafe :
      sig
        val reinit_gram : Token.lexer -> unit;;
        val clear_entry : 'a Entry.e -> unit;;
      end
    ;;
    val extend :
      'a Entry.e -> Gramext.position option ->
        (string option * Gramext.g_assoc option *
           (Gramext.g_symbol list * Gramext.g_action) list)
          list ->
        unit;;
    val delete_rule : 'a Entry.e -> Gramext.g_symbol list -> unit;;
  end
;;
```

Signature type of the functor `Grammar.Make`. The types and functions are almost the same than in generic interface, but:

* Grammars are not values. Functions holding a grammar as parameter do not have this parameter yet.

* The type `parsable` is used in function `parse` instead of the char stream, avoiding the possible loss of tokens.

```
module Make : functor (L : LexerType) -> S;;
```

6.3 Module Pcaml: language grammar, entries and printers.

Hold variables to be set by language syntax extensions. Some of them are provided for quotations management.

```
val gram : Grammar.g;;
val interf : (MLast.sig_item * MLast.loc) list Grammar.Entry.e;;
val implem : (MLast.str_item * MLast.loc) list Grammar.Entry.e;;
val top_phrase : MLast.phrase option Grammar.Entry.e;;
val use_file : MLast.phrase list Grammar.Entry.e;;
val module_type : MLast.module_type Grammar.Entry.e;;
val module_expr : MLast.module_expr Grammar.Entry.e;;
val sig_item : MLast.sig_item Grammar.Entry.e;;
val str_item : MLast.str_item Grammar.Entry.e;;
val expr : MLast.expr Grammar.Entry.e;;
val patt : MLast.patt Grammar.Entry.e;;
val ctyp : MLast.ctyp Grammar.Entry.e;;
val let_binding : (MLast.patt * MLast.expr) Grammar.Entry.e;;
val class_sig_item : MLast.class_sig_item Grammar.Entry.e;;
val class_str_item : MLast.class_str_item Grammar.Entry.e;;
val class_expr : MLast.class_expr Grammar.Entry.e;;
val class_type : MLast.class_type Grammar.Entry.e;;
```

Some grammar and entries of the language, set by `pa_o.cmo` and `pa_r.cmo`.

```
type 'a printer_t = ('a -> unit) ref;;
val print_interf : (MLast.sig_item * MLast.loc) list printer_t;;
val print_implem : (MLast.str_item * MLast.loc) list printer_t;;
```

Some printers, set by `pr_dump.cmo`, `pr_o.cmo` and `pr_r.cmo`.

```
val input_file : string ref;;
```

The file currently being parsed.

```
val output_file : string option ref;;
```

The output file, stdout if None (default)

```
val report_error : exn -> unit;;
```

Prints an error message, using the module `Format`.

```
val quotation_dump_file : string option ref;;
```

`quotation_dump_file` optionally tells the compiler to dump the result of an expander if this result is syntactically incorrect. If `None` (default), this result is not dumped. If `Some fname`, the result is dumped in the file `fname`.

```
val version : string;;
```

The current version of Camlp4.

```
val add_option : string -> Arg.spec -> string -> unit;;
```

Add an option to the command line options.

```
val no_constructors_arity : bool ref;;
```

`True`: dont generate constructor arity.

```
val no_assert : bool ref;;
```

`True`: dont generate assertion checks.

```
val handle_expr_quotation : MLast.loc -> string * string -> MLast.expr;;
val handle_expr_locate : MLast.loc -> int * string -> MLast.expr;;
val handle_patt_quotation : MLast.loc -> string * string -> MLast.patt;;
val handle_patt_locate : MLast.loc -> int * string -> MLast.patt;;
val expr_reloc : (MLast.loc -> MLast.loc) -> int -> MLast.expr -> MLast.expr;;
val patt_reloc : (MLast.loc -> MLast.loc) -> int -> MLast.patt -> MLast.patt;;
```

Allow user to catch exceptions in quotations

```
type err_ctx = Expanding | ParsingResult of (int * int) * string | Locating;;
exception Qerror of string * err_ctx * exn;;
```

for system use

```
val warning : (int * int -> string -> unit) ref;;
val expr_eoi : MLast.expr Grammar.Entry.e;;
val patt_eoi : MLast.patt Grammar.Entry.e;;
val arg_spec_list : unit -> (string * Arg.spec * string) list;;
```

6.4 Module `Plexer`: lexical analyzer

```
val make : unit -> Token.lexer;;
```

Some lexer provided. See the module `Token`. The tokens returned follow the Objective Caml and the Revised lexing rules.

The meaning of the tokens are:

* `("" s)` is the keyword `s`.
* `("LIDENT", s)` is the ident `s` starting with a lowercase letter.
* `("UIDENT", s)` is the ident `s` starting with an uppercase letter.
* `("INT", s)` is an integer constant whose string source is `s`.
* `("FLOAT", s)` is a float constant whose string source is `s`.
* `("STRING", s)` is the string constant `s`.

* ("CHAR", c) is the character constant c.

* ("QUOTATION", "t:s") is a quotation t holding the string s.

* ("ANTIQUOT", "t:s") is an antiquotation t holding the string s.

* ("LOCATE", "i:s") is a location directive at pos i holding s.

* ("EOI", "") is the end of input.

The associated token patterns in the EXTEND statement hold the same names than the first string (constructor name) of the tokens expressions above.

The lexer do not use global (mutable) variables: instantiations of Plexer.make () do not perturb each other.

6.5 Module Quotation: quotation operations

```
type expander =
    ExStr of (bool -> string -> string)
  | ExAst of ((string -> MLast.expr) * (string -> MLast.patt))
;;
```

The type for quotation expanders kind:

* ExStr exp for an expander exp returning a string which can be parsed to create a syntax tree. Its boolean parameter tells whether the quotation is in position of an expression (True) or in position of a pattern (False). Quotations expanders created with this way may work for some particular language syntax, and not for another one (e.g. may work when used with Revised syntax and not when used with Ocaml syntax, and conversely).

* ExAst (expr_exp, patt_exp) for expanders returning directly syntax trees, therefore not necessiting to be parsed afterwards. The function expr_exp is called when the quotation is in position of an expression, and patt_exp when the quotation is in position of a pattern. Quotation expanders created with this way are independant from the language syntax.

```
val add : string -> expander -> unit;;
```

add name exp adds the quotation name associated with the expander exp.

```
val find : string -> expander;;
```

find name returns the expander of the given quotation name.

```
val default : string ref;;
```

default holds the default quotation name.

```
val translate : (string -> string) ref;;
```

function translating quotation names; default = identity

6.6 Module Stdpp: standard definitions

```
exception Exc_located of (int * int) * exn;;
```

Exc_located loc e is an encapsulation of the exception e with the input location loc. To be used in quotation expanders and in grammars to specify some input location for an error. Do not raise this exception directly: rather use the following function raise_with_loc.

```
val raise_with_loc : int * int -> exn -> 'a;;
```

raise_with_loc loc e, if e is already the exception Exc_located, re-raise it, else raise the exception Exc_located loc e.

6.7 Module Token: tokens for grammars

```
type t = string * string;;
type pattern = string * string;;
```

Token and token patterns. Token are build by the lexer. Token patterns come from the EXTEND statement.

The first string is the constructor name (must start with an uppercase character). When it is empty, the second string is supposed to be a keyword.

The second string is the constructor parameter. Empty if it has no parameter.

The way tokens pattern are interpreted to parse tokens is done by the lexer, function tparse below.

```
exception Error of string;;
```

An lexing error exception to be used by lexers.

```
type location_function = int -> int * int;;
```

The type for a function associating a number of a token in a stream (starting from 0) to its source location.

```
type lexer =
  { func : char Stream.t -> t Stream.t * location_function;
    using : pattern -> unit;
    removing : pattern -> unit;
    tparse : pattern -> t Stream.t -> string;
    text : pattern -> string }
;;
```

The type for a lexer used by Camlp4 grammars.

The field `func` is the lexer function. The character stream is the input stream to be lexed. The result is a pair of a tokens stream and a location function for this tokens stream.

The field `using` is a function telling the lexer that the grammar uses this token (pattern). The lexer can check that its constructor is correct, and interpret some kind of tokens as keywords (to record them in its tables). Called by `EXTEND` statements.

The field `removing` is a function telling the lexer that the grammar does not uses the given token (pattern) any more. If the lexer has a notion of "keywords", it can release it from its tables. Called by `DELETE_RULE` statements.

The field `tparse` is a function returning the parsing function associated with each token pattern.

The field `text` returns the name of some token pattern, used in error messages.

Appendix A

Quotations for creating abstract syntax trees

The file "`q_MLast.cmo`" in Camlp4 library directory provides some quotations expanders to create abstract syntax tree nodes using a concrete syntax.

The types generated are defined in the module "`MLast`" provided in the library, but not documented, for the normal usage is to use these quotations.

A.1 General rules

The syntax used in quotations use the revised concrete syntax (chapter 5) with a system of "antiquotations" to insert expressions or patterns of the language in the concrete syntax, to make all possible values of the syntax tree nodes. These antiquotations have the format:

antiquotation `::= $` { *name* `:` } *string* `$`
name `::=` *identifier*

where *string* is any sequence of characters not holding any `$`.

A.2 Quotation `expr`

Syntax tree nodes for language expressions, of type `Ast.expr`. The basic nodes are:

`<:expr< $e1$ . $e2$ >>`: access in records and in modules.

`<:expr< $anti:e$ >>`: location node for Ast antiquotations (temporary node).

`<:expr< $e1$ $e2$ >>`: application.

`<:expr< $e1$ .( $e2$ ) >>`: array access.

`<:expr< [| $list:el$ |] >>`: array.

`<:expr< $e1$ := $e2$ >>`: assignment.

`<:expr< $chr:c$ >>`: character constant.

`<:expr< $flo:s$ >>`: float constant.

`<:expr< for $s$ = $e1$ $to:b$ $e2$ do $list:el$ done >>`: for loop.

`<:expr< fun [ $list:pwel$ ] >>`: function.

`<:expr< if $e1$ then $e2$ else $e3$ >>`: `if` statement.

`<:expr< $int:s$ >>`: integer constant.

`<:expr< let $rec:b$ $list:pel$ in $e$ >>`: `let` statement.

`<:expr< ( $lid:s$ >>`: identifier starting with a lowercase letter.

`<:expr< match $e$ with [ $list:pwel$ ] >>`: `match` statement.

`<:expr< { $list:eel$ } >>`: record.

`<:expr< do $list:el$ return $e$ >>`: sequence.

`<:expr< $e1$ .[ $e2$ ] >>`: string access.

`<:expr< $str:s$ >>`: string constant.

`<:expr< try $e$ with [ $list:pwel$ ] >>`: `try` statement.

`<:expr< ( $list:el$ ) >>`: tuple.

`<:expr< ( $e$ : $t$ ) >>`: type constraint.

`<:expr< $uid:s$ >>`: identifier starring with an uppercase letter.

`<:expr< while $e$ do $list:el$ done >>`: `while` statement.

A.3 Quotation `patt`

Syntax tree nodes for language patterns, of type `Ast.patt`. The basic nodes are:

`<:patt< $p1$ . $p2$ >>`: access in module.

`<:patt< $anti:e$ >>`: location node for Ast antiquotations (temporary node).

`<:patt< ( $p1$ as $p2$ ) >>`: alias.

`<:patt< _ >>`: wildcard.

`<:patt< $p1$ $p2$ >>`: application.

`<:patt< $chr:c$ >>`: character constant.

`<:patt< $int:s$ >>`: integer constant.

`<:patt< $lid:i$ >>`: identifier starting with a lowercase letter.

`<:patt< $p1$ | $p2$ >>`: "or" pattern.

`<:patt< $p1$ .. $p2$ >>`: range.

`<:patt< { $ppl$ } >>`: record.

`<:patt< $str:s$ >>`: string constant.

`<:patt< ( $list:pl$ ) >>`: tuple.

`<:patt< ( $p$ : $t$ ) >>`: type constraint.

`<:patt< $uid:s$ >>`: identifier starting with an uppercase letter.

A.4 Quotation `ctyp`

Syntax tree nodes for language types, of type `Ast.ctyp`. The basic nodes are:

`<:ctyp< $t1$ . $t2$ >>`: access in module.

`<:ctyp< $t1$ as $t2$ >>`: type alias

`<:ctyp< _ >>`: wildcard.

`<:ctyp< $t1$ $t2$ >>`: application.

`<:ctyp< $t1$ -> $t2$ >>`: arrow.

`<:ctyp< $lid:s$ >>`: identfier starting with a lowercase letter.

`<:ctyp< '$s$ >>`: type variable.

`<:ctyp< $t1$ == $t2$ >>`: type manifest.

`<:ctyp< { $list:sbtl$ } >>`: record definition.

`<:ctyp< [ $list:stll$ ] >>`: concrete type definition.

`<:ctyp< ( $list:tl$ ) >>`: tuple.

`<:ctyp< $uid:s$ >>`: identifier starting with an uppercase letter.

A.5 Quotation `sig_item`

Syntax tree nodes for language signature items, of type `Ast.sig_item`. The basic nodes are:

`<:sig_item< declare $list:sil$ end >>`: declare.

`<:sig_item< exception $s$ of $list:tl$ >>`: exception declaration.

`<:sig_item< external $s$ : $t$ = $list:sl$ >>`: declaration of external.

`<:sig_item< module $s$ : $mt$ >>`: module declaration.

`<:sig_item< module type $s$ = $mt$ >>`: module type declaration.

`<:sig_item< open $sl$ >>`: open.

`<:sig_item< type $list:sslt$ >>`: type declaration.

`<:sig_item< value $s$ : $t$ >>`: variable declaration.

A.6 Quotation `str_item`

Syntax tree nodes for language structure items, of type `Ast.str_item`. The basic nodes are:

`<:str_item< declare $list:stl$ end >>`: declare.

`<:str_item< exception $s$ of $list:tl$ >>`: exception declaration.

`<:str_item< $exp:e$ >>`: toplevel expression.

`<:str_item< external $s$ : $t$ = $list:sl$ >>`: declaration of external.

`<:str_item< module $s$ = $me$ >>`: module declaration.

`<:str_item< module type $s$ = $mt$ >>`: module type declaration.

`<:str_item< open $sl$ >>`: open.

`<:str_item< type $list:sslt$ >>`: type declaration.

`<:str_item< value $rec:b$ $list:pel$ >>`: variables definition.

A.7 Quotation module_type

Syntax tree nodes for language module types, of type `Ast.module_type`. The basic nodes are:

`<:module_type< $mt1$ . $mt2$ >>`: access in module.

`<:module_type< $mt1$ $mt2$ >>`: application.

`<:module_type< functor ( $s$ : $mt1$ ) -> $mt2$ >>`: functor.

`<:module_type< $lid:i$ >>`: identifier starting with a lowercase letter.

`<:module_type< sig $list:sil$ end >>`: signature.

`<:module_type< $uid:i$ >>`: identifier starting with an uppercase letter.

`<:module_type< $mt$ with $list:wcl$ >>`: module type with constraint.

A.8 Quotation module_expr

Syntax tree nodes for language module expressions, of type `Ast.module_expr`. The basic nodes are:

`<:module_expr< $me1$ . $me2$ >>`: access in module.

`<:module_expr< $me1$ $me2$ >>`: application.

`<:module_expr< functor ( $s$ : $mt$ ) -> $me$ >>`: functor.

`<:module_expr< struct $list:stl$ end >>`: structure.

`<:module_expr< ( $me$ : $mt$ ) ] >>`: module type constraint.

`<:module_expr< $uid:i$ >>`: identifier starting with an uppercase letter.