Camlp4 - Tutorial

version 3.07

Daniel de Rauglaudre

September 30, 2003

Contents

Chapter 1

Introduction

Camlp4 is a preprocessor for OCaml. As a preprocessor, you can do *syntax extensions* to your OCaml programs. But Camlp4 also provides some other features:

- A system of *grammars*
- A kind of macros called *quotations*
- A revised syntax for OCaml
- A pretty printing system of OCaml programs
- and other: *extensible functions*, *functional streams*, ...

Camlp4 is syntax, syntax, syntax. It uses its own syntax systems to do its own syntax extensions: it is highly bootstrapped. Camlp4 stops at syntax level: it does not know anything about semantic, typing nor code generation (for it, a type definition is just a syntactic thing which starts with "type").

The "p4" in the name "camlp4" stands for the 4 "p" of "Pre-Processor-Pretty-Printer".

1.1 Extending the syntax of OCaml

To start with the beginning, we could try to learn how to make simple syntax extensions to OCaml. If you know the C language, you probably experimented the `define` construction, very easy to use:

```
#define FOO xyzzy
```

and all occurrences of `FOO` in the rest of the program are replaced by `xyzzy`.

In Camlp4, is it not so simple. A syntax extension is not just text replacing: it is an extension of an entry of the grammar of the language, and you need to create syntax trees.

It is therefore necessary 1/ to know what is the grammar system provided by Camlp4 2/ to know how to create syntax trees. It is what we are going to do in this tutorial. Once these points described, we have got the tools to do the syntax extensions of the language.

If you are impatient, and you want to create your syntax extension in the next quarter of an hour, and you don't want to learn all that stuff, you may consider taking the text of an already existing syntax extension and change it for you own needs. A syntax extension is not necessarily a long program (for example, adding the `repeat..until` construction of Pascal takes 6 lines) and you can guess "how it works" and ask the wizards...

Examples are given in chapter 7.

However, if you read this manual, you may be interested on learning the original system of grammars that Camlp4 provides. It can be used for other goals than extending the OCaml language: for your own grammars. This system of grammars is an alternative of yacc: a different approach, but you can describe your language in some identical way.

Just the practical things before (what do I type to experiment?)

1.2 Using Camlp4 as a command and in the toplevel

You must first know that camlp4 is a command. This chapter does not explain all the details of the command and its options: we see that further (chapter 8; you can also use the man pages by typing `"man camlp4"` in your shell).

For the moment, here is a magic incantation to compile a file named `foo.ml`:

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

This command just compiles `foo.ml` as a normal OCaml file, but where the parsing is done by camlp4. The first examples in this documentation (grammars in Camlp4) can be compiled using this command. Otherwise, the examples are given with the correct command to use in order to compile the files.

Another (recommended) better way is to use the OCaml toplevel. In the toplevel, type:

```
#load "camlp4o.cma";;
#load "pa_extend.cmo";;
```

You can type the examples of this documentation in the toplevel. You can also type them in files and use the directive `#use` to include them.

All the examples in this documentation are written in the normal syntax of OCaml, but if you know and prefer the revised syntax provided by Camlp4, change `camlp4o` into `camlp4r` in the `ocamlc` command, or, load `"camlp4r.cma"` instead of `"camlp4o.cma"` in the toplevel.

1.3 Linking applications using Camlp4 libraries

Many examples of this tutorial use some specific Camlp4 libraries. In the toplevel, you don't need to load them because they are in the file `camlp4o.cma`.

To link a standalone application, you need to add the library named `gramlib.cma` of the Camlp4 library directory. The command is:

```
ocamlc -I +camlp4 gramlib.cma <the_files_you_link>
```

1.4 Differences in parsing behavior

Even if you use the normal syntax, there are some small differences in the parsing behavior between the normal `ocamlc` parser (bottom up, LALR parsing) and the `camlp4` parser (top down, recursive descent parsing). These differences appear notably when giving erroneous input. As a trivial example, suppose that you wanted to type

```
(* correct intended input *)
type t = Buf of Buffer.t
       | Str of string
```

Instead of typing the above example, you forgot the second occurrence of the **of** keyword, getting

```
(* file wrongsyntax.ml : wrong input - missing keyword *)
type t = Buf of Buffer.t
       | Str (*missing "of"*) string
```

The `ocamlc` compiler[1] (invoked as `ocamlc -c wrongsyntax.ml`) finds a syntax error on the `string` word; it parses the whole file as a single type declaration and finds a syntax error inside it.
The `camlp4` parser (with ordinary syntax), invoked as `ocamlc -c -pp camlp4o wrongsyntax.ml` don't find any (shallow) syntactic error, but parses the above input as two items:

```
type t = Buf of Buffer.t
       | Str
```

which is a correct type declaration (different from the one intended by the author), followed by a simple expression

```
string
```

which is understood like `let _ = string`, and produces the following message `Unbound value string`

[1]The interactive toplevel `ocaml` has the same behavior, unless you load a different parser.

Chapter 2

Streams and parsers

The first extension provided by Camlp4 is streams and parsers. A stream is a value of the abstract type `'a Stream.t`. It is like a list but where only the first element is available: to read the second element you have to remove the first one from the stream. A parser is a function of type `'a Stream.t -> 'b` which makes a kind of pattern matching inside a stream.

2.1 Streams

2.1.1 Streams constructor

A first way to create a stream is to use the parentheses `[<` and `>]`. There are two kinds of elements: simple elements prefixed by a quote `'` and stream elements not prefixed by it. The stream is composed of the list of the simple and stream elements read in their order.

Examples:

```
[< '3; '1; '4; '1; '5 >]
```

is a stream of these integers.

```
# let s = [< '"hello"; '"world" >];;
# let t = [< '"I"; '"say"; s; '"as"; '"an"; '"example" >];;
```

The above stream `t` is the stream of the strings: "I", "say", "hello", "world", "as", "an", "example".

To look (destructively) inside a stream, you can use the function `Stream.next`, removing the first element from the stream and returning it.

The streams are lazy values. You can create infinite streams, like this one, the stream of all integers:

```
# let rec f n = [< 'n; f (n + 1) >];;
# let s = f 0;;
```

2.1.2 Streams builders

A second way to create streams are the streams builders. They are:

- `Stream.of_string`: returns the character stream of the string given as parameter.
- `Stream.of_channel`: returns the character stream of the input channel.
- `Stream.of_list`: returns the stream of the list elemnts.
- `Stream.from`: given a function, returns the stream composed of the values returned by the function. See the library module `Stream` for details.

Warning: the streams created by these stream builders are much more efficient than the one created by the stream constructors, but they cannot be inserted in them.

2.2 Parsers

The parsers are functions to look inside the streams.

A parser is introduced by the keyword `parser`. It is like the `function` statement, but with stream patterns instead of pattern. For example, the function `Stream.next` introduced in the previous section is defined as:

```
parser [< 'x >] -> x
```

A parser can return with 3 different ways:

- Either by a normal value, indicating that a stream pattern matched the beginning of the stream.
- Or by raising the exception `Stream.Failure`, indicating that no first element of the patterns matched the stream; in this case the stream had not been modified.
- Or by raising the exception `Stream.Error`, indicating that a first element of a pattern matched the input stream, but the rest did not.

Example:

```
# let p = parser [< '3; '1; '4 >] -> "hey";;
# p [< '3; '1; '4 >];;
string : "hey"
# p [< '3; '1; '4; '1; '5; '9 >];;
string : "hey"
# p [< '1; '1; '4 >];;
Exception: Stream.Failure
# p [< '3; '2; '4 >];;
Exception: Stream.Error ""
```

The exceptio `Stream.Error` has a string parameter. You can specify which string, in the parser, after the stream pattern element and the token `??`. For example:

```
# let p =
    parser [< '3; '1 ?? "1 expected"; '4 ?? "4 expected" >] -> "hey"
  ;;
```

Note that the first stream pattern element does not accept this extension since the parser raises `Stream.Failure` if it is not recognized.

A parser can call another parser. You can specify that as a stream pattern element with syntax:

```
pattern = expression
```

Example, with a recursive call:

```
# type tok = If | Then | Else | Let | In | Equal | Ident of int;;
# let rec expr =
    parser
      [< 'If; x = expr; 'Then; y = expr; 'Else; z = expr >] -> "if"
    | [< 'Let; 'Ident x; 'Equal; x = expr; 'In; y = expr >] -> "let"
  ;;
```

Note that a parser is not a system of grammars. You cannot use left recursion:

```
(* BAD EXAMPLE! *)
# let rec expr = parser [< x = expr; 'Equal; y = expr >] -> x;;
```

Which would loop when called, exactly like a function would.

And you cannot start two patterns with the same elements. Only the first one would be applied:

```
(* BAD EXAMPLE! *)
# let rec expr =
    parser
      [< 'If; x = expr; 'Then; y = expr; 'Else; z = expr >] -> "if"
    | [< 'If; x = expr; 'Then; y = expr >] -> "ifnoelse"
  ;;
```

If you want to do that, you need to left factorize your rules:

```
# let rec expr =
    parser
      [< 'If; x = expr; 'Then; y = expr; v = expr_kont >] -> v
  and expr_kont =
    parser
      [< 'Else; z = expr >] -> "if"
    | [< >] -> "ifnoelse"
  ;;
```

or with an anonymous parser:

```
# let rec expr =
    parser
      [< 'If; x = expr; 'Then; y = expr;
         v =
           parser
             [< 'Else; z = expr >] -> "if"
           | [< >] -> "ifnoelse" >] -> v
  ;;
```

Chapter 3

Grammars in Camlp4

A grammar in Camlp4 is a value of type `Grammar.g`. It is created by the function `Grammar.make` which takes a lexer as parameter (see also the functorial interface, another way to create a grammar). Let's ignore for the moment how to create lexers and let's just take a lexer provided in Camlp4, which is the default OCaml lexer. It is in the module `Plexer` and you can create an instantiation by using `Plexer.gmake ()`.

3.1 Grammar, entries: an example

You can create your grammar like this:

```
# let gram = Grammar.gcreate (Plexer.gmake ());;
```

A grammar is composed of `entries`. Entries are values of type `'a Grammar.Entry.e`. The `'a` type parameter represents the type of values which the entry returns. To create an entry, use `Grammar.Entry.create`, which has two parameters: 1/ the associated grammar 2/ a string, the entry name, used for error messages.

An entry is a mutable value. When you create it, it is empty, its type is `'_a Grammar.Entry.e` (not generalized, you cannot create entries returning polymorphic values), the type parameter being set when the entry is extended, showing then the type of its results. Let's take an entry `expr`:

```
# let expr = Grammar.Entry.create gram "expr";;
```

Entries apply to char streams via the function `Grammar.Entry.parse`. If you apply an empty entry (an entry just created, for example), the exception `Stream.Failure` is raised.

To define rules to an entry, you must use the statement `EXTEND`. Note that `EXTEND` is not a syntactic construction in OCaml: it is a syntax extension provided by Camlp4. The syntax of `EXTEND` is:

```
extend-statement ::=
   EXTEND
      list-of-entries-extensions
   END
```

Notice that `EXTEND` is an expression (i.e. not a declaration): it can be evaluated at toplevel, but also inside a function: in this case, the syntax extension takes place when the function is called.

An entry extension has the syntax:

```
entry-extension ::=
   identifier : [ list-of-levels-separated-by-bars ] ;
```

The *identifier* is an entry name. An entry can have one or several levels, representing precedences and associativity.

A level has the syntax:

```
level ::=
   [ list-of-rules-separated-by-bars ]

rule ::=
   list-of-symbols-separated-by-semicolons -> action
```

A *rule* is like a pattern matching case: it can introduce patterns variables which can be used in the *action* part. When the rule is accepted, the action is executed. The type of the *action* is `'a` for an `'a entry` (`'a Grammar.Entry.e`, more precisely).

Let's define our `expr` to parse a simple computation of arithmetic expression with addition, subtraction, multiplication and division, integer constants and parentheses. This can be written:

```
EXTEND
  expr:
    [ [ x = expr; "+"; y = expr -> x + y
      | x = expr; "-"; y = expr -> x - y ]
    | [ x = expr; "*"; y = expr -> x * y
      | x = expr; "/"; y = expr -> x / y ]
    | [ x = INT -> int_of_string x
      | "("; e = expr; ")" -> e ] ]
  ;
END;;
```

The `expr` entry has now three levels. Grammar entries are extensible at any time: you can extend `expr` again to add more constructions. You can add more rules in already existing levels and you can insert new levels. We see that further.

Each level has its own associativity. You can specify left, right or non-associative. By default, the associativity is left. See further.

If you are in the OCaml toplevel and have tested this example, you can see now that `expr` is of type `int entry`, since it returns values of type `int`.

You can try it out:

```
# Grammar.Entry.parse expr (Stream.of_string "2 + 3");;
- : int = 5

# Grammar.Entry.parse expr (Stream.of_string "8 * (5 - 2)");;
- : int = 24
```

And so on.

What happens in case of syntax error? In the general case, the exception `Stream.Error` is raised, enclosed in another exception named `Exc_located` which indicates the location of the error in the stream. Here, the right parenthesis is missing:

```
# Grammar.Entry.parse expr (Stream.of_string "9 / (7 + 1 ");;
Uncaught exception:
Stdpp.Exc_located
 ((11, 12), Stream.Error "')' expected after [expr] (in [expr])").
```

If there are unexpected symbols after a correct expression, it is not a parsing error, the parsing of the stream just stops and the trailing symbols are ignored:

```
# Grammar.Entry.parse expr (Stream.of_string "8 * (5 - 2) 7 foo");;
- : int = 24
```

To ensure that there are no trailing tokens in the input stream, we can create another entry `expr_eoi` expecting an `expr` followed by the end of the input `EOI`:

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

Now:

```
# Grammar.Entry.parse expr_eoi (Stream.of_string "8 * (5 - 2) 7 foo");;
Uncaught exception:
Stdpp.Exc_located
 ((12, 13),
  Stream.Error "end of input expected after [expr] (in [expr_eoi])").
```

3.2 Remark about the lexer

Notice that `"+"`, `"-"`, `"*"`, "/", `EOI` are terminals in this grammar, but they are not specifically predefined in Camlp4 grammar system: it depends on how the associated lexer works. Remember: we used `Plexer.gmake ()` as associated lexer in the present grammar. In another grammar, with another lexer, these terminals might have no meaning.

Before doing the extensions, the statement `EXTEND` first scan all rules and, for each terminal, asks the lexer whether this terminal is correct or not. It uses for that a function named `using` defined in the lexer record type (see interface of module Token). This function can be used also to update a list (or hashtable) of keywords (in the case when there is a notion of keywords, what is not mandatory).

Ok. But it is lexer stuff... We don't need to know about it for the moment. However, it is interesting to know that in the predefined lexer `Plexer.gmake ()`, this function `using` prints an error message and raises an exception if a bad terminal is used in a `EXTEND` statement:

```
# EXTEND expr_eoi: [ [ AAA -> 3 ] ]; END;;
Lexer initialization error.
The constructor "AAA" is not recognized by Plexer
Uncaught exception: Failure "Grammar.extend".

# EXTEND expr: [ [ x = expr; "a+b" -> x + 1 ] ]; END;;
Lexer initialization error.
The token "a+b" does not respect Plexer rules
Uncaught exception: Failure "Grammar.extend".
```

All this details are described in a chapter about lexers, in the reference manual. See also the predefined modules Token and Plexer.

3.3 Labelled levels and associativity

Now, we are going to extend `expr`. But as we defined it, it is not possible to point the entries. To point them, we need labels.

The syntax of a *level* is actually:

```
level ::=
   optional-label optional-associativity
   [ list-of-rules-separated-by-bars ]
```

A label is a string. Any string you choose. The associativity is either `LEFTA`, `RIGHTA` or `NONA`.

Let's write `expr` again (we take a fresh entry, in order to start with an empty entry) with labels and explicit associativity:

```
# let expr = Grammar.Entry.create gram "expr";;
# EXTEND
    expr:
      [ "add" LEFTA
        [ x = expr; "+"; y = expr -> x + y
        | x = expr; "-"; y = expr -> x - y ]
      | "mult" RIGHTA
        [ x = expr; "*"; y = expr -> x * y
        | x = expr; "/"; y = expr -> x / y ]
      | "simple" NONA
        [ x = INT -> int_of_string x
        | "("; e = expr; ")" -> e ] ]
    ;
  END;;
```

By the way, an useful function, especially in the toplevel, is `Grammar.Entry.print`, which displays the contents of an entry (just the rules, in fact):

```
# Grammar.Entry.print expr;;
[ "add" LEFTA
  [ SELF; "+"; SELF
  | SELF; "-"; SELF ]
| "mult" RIGHTA
  [ SELF; "*"; SELF
  | SELF; "/"; SELF ]
| "simple" NONA
  [ "("; SELF; ")"
  | INT ] ]
```

Notice that all `expr` have been replaced by `SELF`: this is the same thing: when an entry calls itself, you can use either its name or the keyword `SELF`. It represents either the current level, the next level or the first level, depending on the associativity and the position of the `SELF` in the rule (current or next level if the `SELF` starts or ends the rule, first level otherwise).

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

```
# EXTEND expr: [ [ x = expr; "plus1plus"; y = expr -> x + 1 + y ] ]; END;;
```

This extended the first level, i.e. the one labelled `"add"`. Type `Grammar.Entry.print expr` to see the result.

You can extend any existing level and insert new levels. Actually, the syntax of an entry extension is:

```
entry-extension ::=
   optional-position
   identifier : [ list-of-levels-separated-by-bars ] ;

position ::=
   FIRST | LAST | BEFORE label | AFTER label | LEVEL label
```

3.4 Extending a level

To extend some specified level, we can use `LEVEL` followed by the label of the level to be extended:

```
# let env = ref [];;
# EXTEND
    expr: LEVEL "simple" [ [ x = LIDENT -> List.assoc x !env ] ];
  END;;
```

You `Grammar.Entry.print expr` again to see the result.

The symbol LIDENT is a constructor defined by our lexer Plexer. It represents an identifier starting by a lowercase character. For details, see the interface of the module Plexer (plexer.mli, given also in the reference manual, chapter libraries).

Just small tests:

```
# Grammar.Entry.parse expr (Stream.of_string "foo + 1");;
Uncaught exception: Stdpp.Exc_located ((3, 4), Not_found)

# env := ("foo", 27) :: !env;;
# Grammar.Entry.parse expr (Stream.of_string "foo + 1");;
- : int = 28
```

3.5 Insert a level

To insert a level, you can use BEFORE or AFTER, relatively to an existing level:

```
# EXTEND
    expr: AFTER "mult"
      [ "power" RIGHTA
        [ x = expr; "**"; y = expr -> int_of_float (float x ** float y) ] ]
    ;
  END;;
```

There is no limit to the number of levels: it is just a list. It is also possible to use FIRST or LAST: they create levels in the beginning or at the end.

3.6 Infix operator

Inside an EXTEND statement you can use antiquotations in places where strings are expected. The antiquotation is some expression between two "dollar" signs. A typical example is a function adding an infix operator "op" at the level "lev":

```
# let add_infix lev op =
    EXTEND
      expr: LEVEL $lev$
        [ [ x = expr; $op$; y = expr -> <:expr< $lid:op$ $x$ $y$ >> ] ]
      ;
    END;;
```

This function can be called when you want to add your infix. The infix becomes automatically a keyword (actually, it depends on the lexer behaviour). It can be used also to define an infix macro in the OCaml grammar (chapter 7).

3.7 Attributed grammars

It is not possible to create attributed grammars, i.e. grammars with parameters (in our terminology, entries with parameters). But entries can return functions. Let us take another version where the entry `expr` returns a function which takes an environment `env`:

```
# let expr = Grammar.Entry.create gram "expr";;
# EXTEND
    expr:
      [ "add" LEFTA
        [ x = expr; "+"; y = expr -> fun env -> x env + y env
        | x = expr; "-"; y = expr -> fun env -> x env - y env ]
      | "mult" RIGHTA
        [ x = expr; "*"; y = expr -> fun env -> x env * y env
        | x = expr; "/"; y = expr -> fun env -> x env / y env ]
      | "simple" NONA
        [ x = INT -> fun env -> int_of_string x
        | x = LIDENT -> fun env -> List.assoc x env
        | "("; e = expr; ")" -> e ] ]
    ;
  END;;
```

To call the entry, we need now to add the environment (a list) as parameter:

```
# Grammar.Entry.parse expr (Stream.of_string "foo + 1") [];;
Uncaught exception: Not_found.
```

(since foo is not in the environment)

```
# Grammar.Entry.parse expr (Stream.of_string "foo + 1") [("foo", 48)];;
- : int = 49
```

We can improve the error message to display the unbound variable name:

```
# EXTEND
    expr: LEVEL "simple"
      [ [ x = LIDENT ->
            fun env ->
              try List.assoc x env with
                Not_found -> failwith ("unbound variable " ^ x) ] ]
    ;
  END;;
```

Notice that there is already a rule `LIDENT` in the level "simple" of `expr`. In this case, the `EXTEND` statement replaces the old version by the new one and displays a warning. To mask this message, one can set the variable `Grammar.warning_verbose` to `false`.

Now, it is more informative:

```
# Grammar.Entry.parse expr (Stream.of_string "foo + 1") [];;
Uncaught exception: Failure "unbound variable foo".
```

3.8 Source location

We can improve our above error system again, by telling the location of the error. In our examples, we tested short texts, it is easy to see the error, but if your grammar becomes very big and treats very big input files, it is very important to know with precision where the error happened in the source.

An useful function for that is `Stdpp.raise_with_loc` taking an input location and an exception as parameters. It raises the exception `Stdpp.Exc_located` already seen some sections above.

We could directly raise this exception `Exc_located` but `raise_with_loc` has the advantage to just re-raise the exception parameter when it is already `Exc_located`, which is useful to propagate exceptions without stacking `Exc_located`.

The input location of a rule is in the variable `loc` always available in the action part:

```
# EXTEND
    expr: LEVEL "simple"
      [ [ x = LIDENT ->
            fun env ->
              try List.assoc x env with
                Not_found ->
                  Stdpp.raise_with_loc loc
                    (Failure ("unbound variable " ^ x)) ] ]
    ;
  END;;

# Grammar.Entry.parse expr (Stream.of_string "3 + foo + 1") [];;
Uncaught exception:
Stdpp.Exc_located ((4, 7), Failure "unbound variable foo").
```

We are going to extend now our entry `expr` with a "let" construction, which can extend the environment.

This is an occasion to introduce the notion of meta symbols in entry rules.

3.9 Meta symbols: lists, options, levels

It is possible to use some meta symbols in rules:

- list of symbols, possibly empty:
  `LIST0 <symbol>`
- list of symbols with a separator, possibly empty:
  `LIST0 <symbol> SEP <symbol>`
- list of symbols with at least one element:
  `LIST1 <symbol>`

- list of symbols with a separator and at least one element:
  `LIST1 <symbol> SEP <symbol>`

- optional symbol:
  `OPT <symbol>`

- levels:
  `[ <rule-list> ]`

Now, we can write a let statement, which calls an non empty list of bindings separated by the keyword "`and`":

```
# let binding = Grammar.Entry.create gram "let_binding";;
# EXTEND
    expr: FIRST
      [ [ "let"; r = LIST1 binding SEP "and"; "in"; e = expr ->
            fun env ->
              let new_env =
                List.fold_right (fun b new_env -> b env :: new_env)
                  r env
              in
              e new_env ] ]
     ;
     binding:
       [ [ p = LIDENT; "="; e = expr -> fun env -> (p, e env) ] ]
     ;
  END;;
```

Let us define an useful function to test our entries:

```
# let apply e s = Grammar.Entry.parse e (Stream.of_string s) [];;
```

Here are some examples:

```
# apply expr "let a = 25 and b = 12 in a + b";;
- : int = 37

# apply expr "let a = 25 and b = a + 5 in a + b";;
Uncaught exception:
Stdpp.Exc_located ((19, 20), Failure "unbound variable a").

# apply expr "let a = 25 in let b = a + 5 in a + b";;
- : int = 55
```

3.10 Local and global entries

We have now three entries: `expr_eoi`, `expr` and `binding`.

In big grammars, we often have to create a lot of small entries, forcing us to define them with `Grammar.Entry.create`. It is not practical for several reasons: 1/ it is tedious 2/ we generally don't need to access all of them directly 3/ the ones which are not eventually extended (which may happen when you perfect your grammar) remain of type `_'a` entry which cause `ocamlc` to fail at end of the module.

To avoid that, it is possible to ask `EXTEND` to automatically define all these small entries. It actually works the wrong way round: you have to define the list of the entries which are globally defined. The other ones (the ones which are "extended" in the statement) are locally defined. Actually, the definition of `EXTEND` is:

```
extend-statement ::=
   EXTEND
      optional-global
      list-of-entries-extensions
   END

global ::=
   GLOBAL : list-of-entries ;
```

Warning: this statement is a little bit complicated to read: `GLOBAL` introduces the list of entries which have been *defined* previously. It means that all other entries in the `EXTEND` statement are automatically locally defined. Therefore, by default, if there is no `GLOBAL`, it must be read: "all entries are global".

In our example, if we want that only `expr_eoi` be defined and be visible, we can add the `GLOBAL` entry with only `expr_eoi`: in this case, `expr` and `binding` are locally defined and therefore not extensible.

To be sure that we don't use the previously defined `expr` and `binding` by quitting the toplevel and entering it again. Don't forget the:

```
#load "camlp4o.cma";;
#load "pa_extend.cmo";;
```

And now:

```
# let gram = Grammar.gcreate (Plexer.gmake ());;
# let expr_eoi = Grammar.Entry.create gram "expr_eoi";;
# EXTEND
    GLOBAL: expr_eoi;
    expr_eoi:
      [ [ e = expr; EOI -> e ] ]
    ;
    expr:
      [ [ "let"; r = LIST1 binding SEP "and"; "in"; e = expr ->
            fun env ->
              let new_env =
```

```
                List.fold_right (fun b new_env -> b env :: new_env)
                  r env
              in
              e new_env ]
      | "add" LEFTA
        [ x = expr; "+"; y = expr -> fun env -> x env + y env
        | x = expr; "-"; y = expr -> fun env -> x env - y env ]
      | "mult" RIGHTA
        [ x = expr; "*"; y = expr -> fun env -> x env * y env
        | x = expr; "/"; y = expr -> fun env -> x env / y env ]
      | "simple" NONA
        [ x = INT -> fun env -> int_of_string x
        | x = LIDENT ->
            (fun env -> try List.assoc x env with
               Not_found ->
                 Stdpp.raise_with_loc loc
                   (Failure ("unbound variable " ^ x)))
        | "("; e = expr; ")" -> e ] ]
    ;
    binding:
      [ [ p = LIDENT; "="; e = expr -> fun env -> (p, e env) ] ]
    ;
  END;;

# let apply e s = Grammar.Entry.parse e (Stream.of_string s) [];;

# apply expr_eoi "let a = 25 and b = 12 in a + b";;
- : int = 37

# apply expr_eoi "let a = 25 and b = 12 in a + b foo bar";;
Uncaught exception:
Stdpp.Exc_located
  ((31, 34),
     Stream.Error "end of input expected after [expr] (in [expr_eoi])")
```

3.11 Deleting a rule

To delete a rule, use the statement `DELETE_RULE`. Its syntax is:

```
delete-rule ::=
   DELETE_RULE entry : list-of-symbols-separated-by-semicolons END
```

It deletes the first rule found in the levels of `entry` matching the list of symbols.

For example, in the above example, you can delete the “addition” rule of the entry `expr`, by typing:

```
# DELETE_RULE expr: SELF; "+"; SELF END;;
```

3.12 Parsed language

The grammar entries levels are just improved streams parsers. Streams parsers use recursive descendant parsing (something close to LL(1) but actually a little less powerful). The improvements are:

- left factorization: you can e.g. then write a rule "if..then" and a rule "if..then..else" starting with the same symbols, it works.

- automatic treatment of associativity and level precedence, as we just saw.

There is no left factorization between different entries, or in different levels in the same entry.

This is often a problem. This example does not work:

x ::= y | z
y ::= `A B | ...`
z ::= `A C | ...`

The input `"A C"` raises `Stream.Error`. There is no simple solution to this problem, but there is a solution (not very clean, actually): create a entry from a parser (it is possible via the function `Grammar.Entry.of_parser`). This parser must scan the stream using `Stream.npeek` until the number of necessary tokens allows to make the difference; return unit or raise `Stream.Failure` according to the cases.

3.13 Functorial interface

There is another way to create a grammar: using the functional interface. See the module `Grammar`. In this case, grammars are no more values, but modules. The extension of a grammar is done by the keyword `GEXTEND` followed by the grammar module name.

The differences are the following:

- OCaml normal typing ensures that entries call only entries of the same grammar, because each grammar module defines its own entry type. In the non-functorial interface, this is checked at execution time.

- The input of the function `Entry.parse` is a value of type `parsable`, instead of the direct character stream. You must create a parsable value from a character stream by the function `parsable`. This ensures that there is no lack of tokens when calling `Entry.parse` several times. In the normal interface, when calling `Grammar.Entry.parse` with a char stream, the grammar system may ask the lexer for a token which may not be used (depending on the entries rules), but lost when restarting from the character stream.

Use the normal interface or the functorial interface is a question of personal taste.

3.14 Writing a lexer

If you want to write your own lexer, a simple solution is to take the sources of the provided lexer (plexer.ml) and make your changes in it.

Otherwise you can read the section "writing a lexer" in the reference manual.

3.15 Grammar for OCaml syntax

The command camlp4 uses to parse OCaml files: for that, it uses extensible grammars described in the present chapter (a grammar with extensible entries). The main entries of the grammar of OCaml are accessible and therefore extensible: for expressions, for patterns, for structures, signatures, and so on. All are defined in the provided module `Pcaml`. See chapter 7, the reference manual, chapter of library modules or look at the interface `pcaml.mli`.

But we are not yet ready to write syntax extensions for OCaml: we first need to create syntax trees which are returned by these entries.

The way to create OCaml syntax trees is explained in the following chapters, the first one being about another feature of Camlp4: the `quotations`.

Chapter 4

Quotations

Quotations are expressions or patterns enclosed by special parentheses: `<:id<` and `>>` (`id` is a quotation identifier). They exist also enclosed by `<<` and `>>`.

Examples of quotations:

```
<:expr< let a = b in c >>
<< [x](x y) >>
<:myquot< quotations can be any text >>
```

The contents of quotations are not lexed: quotations themselves are tokens, exactly like strings. Therefore like the contents of strings, their contents do not have to respect any special lexing rule.

4.1 OCaml syntax extensions

In the previous chapter, we saw the grammar system of Camlp4. This was a first step in the way to be able to write syntax extensions in OCaml.

The second step is: how to make OCaml syntax trees nodes? The immediate answer is: use the module defining them. Indeed, this module exist: its name is `MLast`. You can then try do understand it or... you can use quotations.

Let us suppose we want to generate the syntax tree of the OCaml expression: `"let a = b in c"`. The version using the module `MLast` directly is:

```
MLast.ExLet
  (loc, false, [MLast.PaLid (loc, "a"), MLast.ExLid (loc, "b")],
   MLast.ExLid (loc, "c"))
```

Not so complicated, perhaps. But you need directions for use detailing all tree nodes, their parameters, their usages. If you are courageous, you may try to look inside the code of Camlp4 to see how they are used.

But the quotation system of Camlp4 provides a handy way to represent these trees. In this system, if the right file is loaded, you are able to write the above example as:

```
<:expr< let a = b in c >>
```

Simpler, isn't it? Everything inside is treated at compile time by Camlp4 which generates exactly the same code than the above "long" version.

Let us look in details at the Camlp4 quotation system, then. It can be used for these tree notes of OCaml syntax, but actually for any other type. You can define your own quotations, using any syntax you choose.

4.2 Camlp4 quotation system

The quotations contents are not lexed, as said above, but they are even so treated at parse time. Actually they are not lexed by the OCaml lexer (of Camlp4), but they are analyzed by other functions, which are called "quotation expanders". These expanders just take a string as parameter (the contents of the quotation) and return a piece of OCaml program.

There are two kinds of quotations expanders, which do the same things: but one version is easy to use, although not general, and the other one is general but need to have a knowledge of... OCaml syntax trees quotations (them again).

Well... this second version needs the knowledge of quotations to learn quotations... and the OCaml syntax trees quotations are explained in the next chapter.

Ok, since we are stuck in a mutually recursive documentation, let us start rather with the "simple" quotations, the ones "easy to use". This will explain you how work quotation expanders. In a second step, we explain the general quotations which need `MLast` quotations to be defined.

4.3 Example: defining constants

The simple version of quotation expanders just return strings. These returned strings are a piece of code, in concrete syntax, like source code. And they *are* source code, in a meaning that they need to be parsed after the quotation expansion.

Let us take a small example. We want to define constants by their name (this example is almost the equivalent of the `#define` in C for the case when a simple constant is defined).

You are going to create our examples in the toplevel by typing (under the toplevel):

```
#load "camlp4o.cma";;
```

Now, we can type this:

```
# let expand _ s =
    match s with
      "PI" -> "3.14159"
    | "goban" -> "19*19"
    | "chess" -> "8*8"
```

```
    | "ZERO" -> "0"
    | "ONE" -> "1"
    | _ -> "\"" ^ s ^ "\""
  ;;
```

Let us call the quotation "foo". We can associate the quotation "foo" to the above expander "expand" by typing:

```
# Quotation.add "foo" (Quotation.ExStr expand);;
```

We can experiment the new quotation immediately:

```
# <:foo<PI>>;;
- : float = 3.14159
# <:foo< hello, world >>;;
- : string = " hello, world "
# <:foo<ONE>> + <:foo<ONE>>;;
- : int = 2
# let rec fact x =
    if x = <:foo<ZERO>> then <:foo<ONE>> else x * fact (x - 1)
  ;;
val fact : int -> int = <fun>
```

And so on. But quotations can also be used as patterns:

```
# let rec fib =
    function
      <:foo<ZERO>> | <:foo<ONE>> -> 1
    | n -> fib (n - 1) + fib (n - 2)
  ;;
val fib : int -> int = <fun>
```

You can remark that a specific quotation has not a specific type: its type depends on what the quotation generates: it this case, it can be a float number, an integer, a string. Notice also that the spaces inside quotations are significant: here the expander do not strip them, and therefore:

```
# <:foo< PI >>;;
```

since `" PI "` with two spaces around is not matched by `"PI"` without spaces, doesn't return the value of `PI` but:

```
- : string = " PI "
```

If we want that both cases return the value of PI, we should have written a more subtle quotation expander. A quotation expander can use any parsing technology: string pattern matching (in our example), stream parsers, ocamllex, ocamlyacc, Camlp4 grammars... What is important is that they take a string as parameter and return a piece of program.

4.4 Quotations and the compiler

We just saw an example of quotation in the toplevel: but in this context, compilation level and program level are mixed. When using the compiler `ocamlc`, the quotation system requires that the two levels be separated:

- The quotation expander and the statement `Quotation.add` are directives to the compiler.
- The usage of quotations are in the user program.

The "compiler part" must be previously compiled. To test the above example, we have to copy the text of the function `expand` and the call to `Quotation.add` in a file, e.g. `foo.ml`, which must be compiled like this:

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

This creates an object file named `foo.cmo`. *In Camlp4, all syntax extensions are done through OCaml object files.* The preprocessor `camlp4o` takes a list of object files as first arguments and load them. Let us write a file `fib.ml`:

```
(* file fib.ml *)
let rec fib =
  function
    <:foo<ZERO>> | <:foo<ONE>> -> 1
  | n -> fib (n - 1) + fib (n - 2)
;;
```

As a first remark, we can see that the normal OCaml compiler does not know quotations:

```
$ ocamlc -c fib.ml
File "fib.ml", line 4, characters 4-6:
Syntax error
```

But Camlp4 does...

```
$ ocamlc -pp camlp4o -c fib.ml
File "fib.ml", line 4, characters 4-16:
While expanding quotation "foo":
Uncaught exception: Not_found
Preprocessing error
```

... providing the quotation expander object file is given as parameter (it must be written `./foo.cmo` because `camlp4` does not have the current directory in its default research path):

```
$ ocamlc -pp "camlp4o ./foo.cmo" -c fib.ml
```

4.5 Pretty printing the result

How to be sure that the quotations are correctly expanded? If you are perfecting a quotation expander or if you already have got a quotation expander, and you want to see the results, you can ask `camlp4` to pretty print the result.

For that, use `camlp4o` as command with the predefined printing kit, named `"pr_o.cmo"`:

```
$ camlp4o ./foo.cmo pr_o.cmo fib.ml
(* file fib.ml *)
let rec fib =
  function
    0 | 1 -> 1
  | n -> fib (n - 1) + fib (n - 2)
;;
```

The quotations have been replaced by their value.

4.6 Quotations returning syntax trees

Our quotation expander, the function "expand", returns strings. Internally, when `camlp4` encounters a quotation "foo" in the program text, it calls this function and get the resulted string. This string is parsed with the grammar entry "expr" (expressions) or "patt" (pattern).

But this has the following drawbacks:

1/ It needs a new parsing phase (which takes time, not much but it is too bad).
2/ If the expander is bad written, this resulting string may be syntactically incorrect, and it is difficult to debug (however see the option `-QD` of camlp4).
3/ It is dependent from the enclosing syntax: the same expander may work e.g. in revised syntax but not in normal syntax.

To illustrate the point 2/, try to type this:

```
# <:foo< to"to >>;;
```

The result is this strange message:

```
# <:foo< to"to >>;;
  ^^^^^^^^^^^^^^^
While parsing result of quotation "foo":
(consider setting variable Pcaml.quotation_dump_file)
Parse error: end of input expected after [expr] (in [expression])
```

It is because our quotation expander was too simple: it created a string containing: a double quote, the contents of the quotation, another double quote, i.e:

```
 " to"to "
```

The parser then fails with this input. Debugging this can be sometimes complicated, especially if the expander does not pretty print its results, or add a lot of redundant parentheses, etc. Here, the solution would have been to use "String.escaped s" instead of "s" in the expander.

To avoid that, and all the other drawbacks, there is the other quotation system, the one where expanders return abstract syntax trees. In "expand", instead of returning the string "3.14159", we may want to say "the syntax tree representation of the float number 3.14159". In this case, it does not need another parsing phase, no risk of parse error, and it is independent from the enclosing syntax.

The way to create OCaml syntax trees is explained in chapter 6. They can be written in quotations, using the syntax extension kit named q_MLast.cmo.

The same quotation expander in our file foo.ml could be written:

```
(* file foo.ml *)
let loc = (0, 0);;
let expand_expr s =
  match s with
    "PI" -> <:expr< 3.14159 >>
  | "goban" -> <:expr< 19 * 19 >>
  | "chess" -> <:expr< 8 * 8 >>
  | "ZERO" -> <:expr< 0 >>
  | "ONE" -> <:expr< 1 >>
  | _ -> <:expr< $str:s$ >>
;;
let expand_patt s =
  match s with
    "PI" -> <:patt< 3.14159 >>
  | "ZERO" -> <:patt< 0 >>
  | "ONE" -> <:patt< 1 >>
  | _ -> <:patt< $str:s$ >>
;;
Quotation.add "foo" (Quotation.ExAst (expand_expr, expand_patt))
```

This time we used ExAst instead of ExStr. This constructor needs two expanders: one for the quotations in position of expression and one for the quotations in position of pattern. Notice that the cases "goban" and "chess" are not in the pattern version since 19*19 and 8*8 are not correct patterns.

The compilation of foo.ml needs the quotation expander kit q_MLast.cmo:

```
$ ocamlc -pp "camlp4o q_MLast.cmo" -I +camlp4 -c foo.ml
```

This creates an object file "foo.cmo" which can be used to compile "fib.ml".

Just for curiosity, you can pretty print the expander itself, using the pretty printing kit pr_o.cmo. Type:

```
$ camlp4o q_MLast.cmo pr_o.cmo foo.ml
```

4.7 Example: lambda terms

We can now take a bigger example, bigger than just creating constants. We want to manipulate lambda terms. A lambda term can be defined by the following type:

```
type term =
    Var of string
  | Func of string * term
  | Appl of term * term
;;
```

The first case, `Var`, represents variables.

The second case, `Func`, represents functions. Its first parameter is the function parameter and its second parameter the function body. We write that in concrete syntax `[parameter]body`.

The third case, `App`, represents an application of two lambda terms. We write that in concrete syntax `(term1 term2)`.

But, for the moment, we just defined a type `term`, and we can just write these terms using the constructors. Here is an example:

```
let id = Func ("x", Var "x")
let k = Func ("x", Func ("y", Var "x"))
let s =
  Func ("x", Func ("y", Func ("z",
    Appl (Appl (Var "x", Var "y"), Appl (Var "x", Var "z")))))
let delta = Func ("x", Appl (Var "x", Var "x"))
let omega = Appl (delta, delta)
```

A nice quotation expander would allow us to use concrete syntax. The same piece of program could look like this, which is more readable:

```
let id = << [x]x >>
let k = << [x][y]x >>
let s = << [x][y][z]((x y) (x z)) >>
let delta = << [x](x x) >>
let omega = << (^delta ^delta) >>
```

Let us write the corresponding quotation expander, then.

Here, the contents of our quotations is too complicated to be parsed just by string pattern matching. We could use a stream parser, but the simplest way is to use grammars.

No need to write a lexer, the default lexer Plexer provided in the Camlp4 library fits. Using our knowledge (previous chapter) about Camlp4 grammars, here is a quotation expander for the lambda terms: (file named `q_term.ml`):

```
let gram = Grammar.gcreate (Plexer.gmake ());;
let term_eoi = Grammar.Entry.create gram "term";;
let term = Grammar.Entry.create gram "term";;
EXTEND
   term_eoi: [ [ x = term; EOI -> x ] ];
   term:
      [ [ "["; x = LIDENT; "]"; t = term -> <:expr< Func $str:x$ $t$ >>
        | "("; t1 = term; t2 = term; ")" -> <:expr< Appl $t1$ $t2$ >>
        | x = LIDENT -> <:expr< Var $str:x$ >> ] ]
   ;
END;;
let term_exp s = Grammar.Entry.parse term_eoi (Stream.of_string s);;
let term_pat s = failwith "not implemented term_pat";;
Quotation.add "term" (Quotation.ExAst (term_exp, term_pat));;
Quotation.default := "term";;
```

Several remarks about the text of this quotation expander:

- Inside the expr quotations (action parts in the entry `term`), you see things between dollar signs: these are *antiquotations*. Antiquotations allow to insert piece of expressions (or patterns) inside quotations. We see that very soon.
- Still in the quotations, the constructors `Func` and `Appl` are applied without parentheses: it is because inside the predefined quotations for OCaml syntax trees, the "revised syntax" is used: close to the normal OCaml syntax, this syntax has however some small differences: it is the case for constructors application: in revised syntax, you have to write `Func x y` instead of `Func (x, y)`. See chapter 5.
- The final statement `Quotation.default := "term"` tells camlp4 that `"term"` is the default quotation, allowing us to use them between `<<` and `>>`.
- Notice that you don't need to define the type `term` in this file: the constructors are used only in the syntax trees which internally represent them in strings.
- The pattern version is not written, we shall add it in the final version.

To compile `q_term.ml`, we need:

1/ the `pa_extend.cmo` syntax extension, for the `EXTEND` statement
2/ the `q_MLast.cmo` quotation extension, for the OCaml syntax trees quotations.

The compilation must then be done by the command:

```
$ ocamlc -pp "camlp4o q_MLast.cmo pa_extend.cmo" -I +camlp4 \
     -c q_term.ml
```

Now we can use the lambda term quotation `q_term.cmo` we just created. Under the toplevel, you can load it (after having loaded `camlp4o.cma`) and use the term quotation directly. But, at this level, we need to have defined the type `term`, otherwise:

```
# let id = << [x]x >>;;
             ^^^^^^^^^^

Unbound constructor Func
```

Ok, enter the definition of the type `term` in the toplevel. Then:

```
# let id = << [x]x >>;;
val id : term = Func ("x", Var "x")

# let k = << [x][y]x >>;;
val k : term = Func ("x", Func ("y", Var "x"))

# let s = << [x][y][z]((x y) (x z)) >>;;
val s : term =
  Func
   ("x",
    Func
     ("y",
      Func
       ("z", Appl (Appl (Var "x", Var "y"), Appl (Var "x", Var "z")))))

# let delta = << [x](x x) >>;;
val delta : term = Func ("x", Appl (Var "x", Var "x"))
```

The definition of `omega` given in the initial example is a special case that we are going to see in the next section. For the moment, it just answers:

```
# let omega = << (^delta ^delta) >>;;
                  ^

While expanding quotation "term":
Parse error: [term] expected after '(' (in [term])
```

We can remark that the location of the syntax error is correct: this is due to the grammar system: in case of syntax error, the error exception is enclosed with the exception `exc_located` which transmits the error location. Receiving this error, the quotation expansion machinery just has to add the location of the quotation to be able to display the error location correctly in the input text.

Let us see now the case of this definition of the variable `omega`. It can be resolved with antiquotations.

4.8 Antiquotations

Antiquotation is a way to insert code inside quotations. Unlike quotations, antiquotations are not a predefined notion of Camlp4: it is just a programming technique.

In our example, we would like `omega` to be "the application of delta to itself". But when we say "delta", we don't mean "a variable delta" in the context of a lambda term (which would be `Var "delta"`), but "the value of the variable delta" previously defined. We want to insert its value (twice, in this example) to create the new lambda term.

In the very initial version, we had written:

```
let omega = Appl (delta, delta)
```

Ok, we could use that, it is still correct, but as we have a system of quotations, we would like to represent that as concrete syntax, with an application (the two terms between parentheses).

In our concrete syntax, we need to add a specific case to specify "a value of the enclosing environment". Here we chose the caret sign `^` followed by an identifier.

We then have to add a grammar rule which says: "if caret sign followed by and identifier, return the syntax tree of the identifier itself considered as a variable". The rule can be written inside the `EXTEND` statement:

```
  "^"; x = LIDENT -> <:expr< $lid:x$ >>
```

This right `"expr"` quotation represents the OCaml syntax tree of a variable whose name is `"x"`. See chapter 6. Adding this rule in the quotation expander, recompiling it, we can now test in the toplevel:

```
# let delta = << [x](x x) >>;;
val delta : term = Func ("x", Appl (Var "x", Var "x"))

# let omega = << (^delta ^delta) >>;;
val omega : term =
  Appl
    (Func ("x", Appl (Var "x", Var "x")),
     Func ("x", Appl (Var "x", Var "x")))
```

4.9 Locations in antiquotations

Warning: this section is a little bit subtle, to resolve a specific problem. You may skip it, if you find it too complicated or not interesting in a first step.

It is about the location of possible semantic error. By default the whole quotation is underlined:

```
# let omega = << (^delta ^xxx) >>;;
              ^^^^^^^^^^^^^^^^^^^
Unbound value xxx
```

However the variable `xxx` has a location, in the quotation. And the grammar system is supposed to take care of locations, via a variable named `loc` transmitted from the grammar rule to the action part, which is used by the quotations of OCaml syntax trees. But precise locations has been lost. Why?

It is because the Camlp4 quotation machinery ignores if the syntax trees you built have correct locations. It just receives a syntax tree, but does not know which technique you used[1]. The quotation expander might have inserted eccentric locations: in this case, in case of semantic error, the error location could be anywhere in the input text, inside the quotation but at a wrong place, or outside the quotation. even possibly outside the text.

To avoid a possible problem, the Camlp4 quotation machinery wisely scans the resulting syntax tree and put the location of the whole quotation in all nodes (erasing the old ones): better being sure that all semantic error underline the whole quotation rather than risk that the errors messages be anywhere in the input text, or referring bad parts of the quotation.

By default, thus, the quotation machinery does not trust the programmer of the quotation expander. There is however a way to tell it that the location is correct: you can specify in the resulting tree that some specific part contains correct locations.

This can be done by creating an "antiquotation" node. It can be written, for an expression `e` or a pattern `p`:

```
<:expr< $anti:e$ >>
<:patt< $anti:p$ >>
```

Here, in the rule:

```
"^"; x = LIDENT -> <:expr< $lid:x$ >>
```

the right part is just the tree node for the identifier x. It contains the location of the identifier (including the caret, actually). But as it is not enclosed by an "antiquotation" node, the quotation machinery will erase the location and replace it by the location of the whole quotation.

We are then going to use such an node. But we have to remark first that the location of the rule is not exactly what we want, since it includes the caret sign and the possible spaces between it and the identifier. To be very correct, we should use another grammar entry, e.g. `quotation`, holding just one case `LIDENT` and create the antiquotation there where we are sure that the `"loc"` variable represents the location of the `LIDENT`.

The subtree under the antiquotation node is the tree to represent the variable `x`. We cannot use `<:expr< $lid:x$ >>` directly, because it would take `loc` which is the location of the antiquotation itself, and we need the location of the variable *relative* to the beginning of the antiquotation. Its location is then:

```
(0, String.length x)
```

[1]The quotation machinery receives the quotation expander via the function Quotation.add: for it, the expander is just a function; it does not know how it is implemented.

With all these remarks, the line for the antiquotation must then be:

```
"^"; x = antiquot -> x
```

and the `antiquot` entry:

```
antiquot:
  [ [ x = LIDENT ->
        let ast =
          let loc = (0, String.length x) in
          <:expr< $lid:x$ >>
        in
        <:expr< $anti:ast$ >> ] ]
;
```

The syntax of your antiquotations is the one you want, providing you can isolate it from the rest of your quotation. In this example, the antiquotation is introduced by `"^"` and the antiquotation is just an identifier.

But you may want that your antiquotations are between some kind of "parentheses". In this case, you need a way to build the syntax tree of the antiquotation. For that, use:

```
Grammar.Entry.parse Pcaml.expr_eoi (Stream.of_string s)
```

where `s` is the antiquotation string: this will create your antiquotation subtree (if your quotation is in a pattern, instead of an expression, use `patt_eoi` instead of `expr_eoi`).

For example, our predefined antiquotations building syntax trees (see chapter 6) are between two "dollar" signs, allowing users to write things like that:

```
<:sig_item< value $x ^ string_of_int n$ : unit -> unit >>
```

To build the syntax tree of this antiquotation, our quotation `‘‘sig_item’’` applies the above call to the antiquotation string, where `s` is the contents of the antiquotion, i.e.:

```
"x ^ string_of_int n"
```

Notice that the antiquotation subtree will, by default, inherit the location of the whole quotation. If you want it to have its own location (which is very interesting in case of typing errors), don't forget to enclose it with the antiquotation node; you also need to enclose this call with a `try...with` in case of syntax error in the antiquotation string and recompute the error location:

```
let ast =
  try Grammar.Entry.parse Pcaml.expr_eoi (Stream.of_string s) with
    Stdpp.Exc_located (bp, ep) exc ->
      raise_with_loc (fst loc + bp, fst loc + ep) exc
in
<:expr< $anti:ast$ >>
```

4.10 Example: lambda terms, finished

Here is now our quotation expander for lambda terms, including the antiquotation location system of the previous section and the pattern version:

```
let gram = Grammar.gcreate (Plexer.gmake ());;
let term_exp_eoi = Grammar.Entry.create gram "term";;
let term_pat_eoi = Grammar.Entry.create gram "term";;
EXTEND
  GLOBAL: term_exp_eoi term_pat_eoi;
  term_exp_eoi: [ [ x = term_exp; EOI -> x ] ];
  term_exp:
    [ [ "["; x = LIDENT; "]"; t = term_exp -> <:expr< Func $str:x$ $t$ >>
      | "("; t1 = term_exp; t2 = term_exp; ")" -> <:expr< Appl $t1$ $t2$ >>
      | x = LIDENT -> <:expr< Var $str:x$ >>
      | "^"; x = exp_antiquot -> x ] ]
  ;
  exp_antiquot:
    [ [ x = LIDENT ->
          let ast = let loc = (0, String.length x) in <:expr< $lid:x$ >> in
          <:expr< $anti:ast$ >> ] ]
  ;
  term_pat_eoi: [ [ x = term_pat; EOI -> x ] ];
  term_pat:
    [ [ "["; x = LIDENT; "]"; t = term_pat -> <:patt< Func $str:x$ $t$ >>
      | "("; t1 = term_pat; t2 = term_pat; ")" -> <:patt< Appl $t1$ $t2$ >>
      | x = LIDENT -> <:patt< Var $str:x$ >>
      | "^"; x = pat_antiquot -> x ] ]
  ;
  pat_antiquot:
    [ [ x = LIDENT ->
          let ast = let loc = (0, String.length x) in <:patt< $lid:x$ >> in
          <:patt< $anti:ast$ >> ] ]
  ;
END;;
let term_exp s = Grammar.Entry.parse term_exp_eoi (Stream.of_string s);;
let term_pat s = Grammar.Entry.parse term_pat_eoi (Stream.of_string s);;
Quotation.add "term" (Quotation.ExAst (term_exp, term_pat));;
Quotation.default := "term";;
```

After compilation of this file, `q_term.ml`, some experiments in the toplevel (after having loaded `camlp4o.cma` and `q_term.cmo` and defined the type `term`):

```
# let omega = << (^delta ^xxx) >>;;
                  ^^^^^
Unbound value delta
```

```
# let delta = << [x](x x) >>;;
val delta : term = Func ("x", Appl (Var "x", Var "x"))

# let omega = << (^delta ^delta) >>;;
val omega : term =
  Appl
   (Func ("x", Appl (Var "x", Var "x")),
    Func ("x", Appl (Var "x", Var "x")))

# match omega with << (^a ^b) >> -> a | x -> x;;
- : term = Func ("x", Appl (Var "x", Var "x"))
```

Many improvements could be done, for example, to be able to accept the any pattern "_" (any) in order to be able to write this last example:

```
match omega with << (^a ^_) >> -> a | x -> x;;
```

4.11 Conclusion

You know the main points about quotations. In the road to be able to make syntax extension in OCaml, we have advanced to the second step: the general quotation system of Camlp4. Now, we could look in detail at the specific predefined quotations for OCaml syntax trees. We already used some of them. However, as they use the "revised syntax", we must introduce it before.

Chapter 5

The revised syntax

The revised syntax is an alternative syntax for OCaml. Its purposes are 1/ fix some problems of the normal syntax (unclosed constructions sometimes introducing ambiguities, constructors arity, end of top level phrases and structure items, etc) 2/ avoid unjustified double constructions (`":="` vs "`<-`", "fun" vs "function", "begin..end" vs parentheses) or concepts (types and types declarations) 3/ bring some ideas (lists, types). In a word, propose a syntax which be more logical, simpler, more consistent and easier to parse and to pretty print.

The revised syntax, being few used, is less constrained by the history than the normal one, and can try to answer the question: "how things should be done" instead of "how to remain compatible with old versions".

Other motivations are: 1/ show that syntax is just a "shell" of the language: you can change it without modifying the background 2/ experiment right to the end the ability of Camlp4 of doing syntax extensions.

It is a syntax of the complete language, therefore it can be used for all OCaml programs: by the way, Camlp4 is itself completely written in that syntax. Notice that it is not a constraint: it is always possible to convert from and to the normal syntax, using the pretty print facilities of Camlp4.

Remark: syntax in programming languages is much a question of personal taste. This syntax represents mine, with some ideas taken here and there. Some choices may seem arbitrary (other solutions are possible), but I tried to keep some consistency, and without being too far from the normal syntax: I guess that it is possible to understand a program written in revised syntax even without having read this chapter.

Most of the constructions in revised syntax are therefore the same than in the normal syntax. This chapter presents only the differences, and the motivations of them.

The quotations for OCaml syntax trees, which we shall see in next chapter, use the revised syntax.

5.1 Practical points

To compile the file `foo.ml` written in revised syntax, use:

```
$ ocamlc -pp camlp4r foo.ml
```

To use the revised syntax in the toplevel, do:

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

5.2 Phrases

- In revised syntax, simple semicolons end the items of structures, signatures and objects. These semicolons are *mandatory*. The double semicolon is no more a token. There is no ambiguity with the sequence, which has a special construction (see further).
- 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;`

Motivation of the simple semicolon

The double semicolon in OCaml exists for historical reasons: the first parsers were driven by the tokens, not by the rules: all constructions needed to have a specific token.

But because of the introduction of modules in OCaml, the double semicolon, which was mandatory in Caml Light to end sentences, became optional: the reason is that in OCaml, a "phrase" and a "structure item" are actually the same notion. The problem is that the double semicolon is associated with the idea of "terminating" something: for a phrase, it is exact, but not for a structure item inside a structure, since other structure items and the keyword "end" follow.

That choice of letting the double semicolon be optional in normal syntax has introduced several problems:

- A structure item is actually ended by the beginning of the next structure item; it means that all structure items must start with a keyword; otherwise there is an ambiguity. For example, you cannot write:

  ```
  print_string "hello, world"
  print_newline ()
  ```

  because it is interpreted as a call to `print_string` with 3 parameters (and typing error). The advocated solution is to write:

  ```
  let _ = print_string "hello, world"
  let _ = print_newline ()
  ```

  Mmm....

- But this solution does not work interactively: in the toplevel, you cannot ask people to type the beginning of the next sentence to see the result of the current one. Therefore the double semicolon still remains! The property that we write in the toplevel like in source files has been lost.

- In structures and objects, the fact that you don't end the structure items and object items make the programs more difficult to read. If you write a short object or structure item in one only line, it is very difficult to see where the items start and end.

My opinion is that the structure items should end with a token in a context where there is never need to read another token. This ensures a correct behavior in the interactive toplevel. The fact that the sequence is closed, in the revised syntax, frees the simple semicolon. And a simple semicolon is perfectly acceptable inside structures and objects, to end their item, the same way they close a record item. In the revised syntax, this ending semicolon is mandatory.

It is easier to treat a language whose all phrases end with a token: at end of the sentences, the characters and the tokens streams are synchronized (no need to read an extra token to be sure that the phrase is ended). This property can bring simplifications in other treatments (extraction of comments or code for documentation, indentation, editors modes, interactive tools).

Motivation of "value"

The choice of having a different keyword `value` instead of `let`, for a toplevel value definition, is to mark the difference with the `let..in` construct. At toplevel, to see if it is a `let` or or `let..in`, we have to look at the end of the let binding.

In the abstract syntax tree, `let` and `let...in` are very different: they do not even have the same type: `let` is a structure item, while `let...in` is an expression. This deserves to be more visible in the concrete syntax.

Why not `val` instead of `value`? It is to be coherent with the other declarations `type` and `exception`, which are not abbreviations: we don't write `typ` for type declarations, nor `exc` for exception declarations.

5.3 Imperative constructions

- The sequence is introduced by the keyword "`do`" followed by "`{`" and terminated by "`}`" (it is possible to put a semicolon after the last expression):

OCaml	Revised
`e1; e2; e3; e4`	`do { e1; e2; e3; e4 }`

- The body of "`for`" and "`while`" has the same syntax:

OCaml	Revised

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

- The "lets" apply up to the end of the sequences.

Motivation of "do" and braces

First, the sequence needed to be closed. For the reason of the previous section (toplevel phrases), but also because there are too many ambiguities with other constructions. For example in the list:

```
[ a; b; c ]
```

We know that it is the list of `a`, `b` and `c`. But it could be interpreted as a list one element, the sequence `"a; b; c"`. In the grammar, it supposes that list items are not "top" expressions (expressions of the first level of the "expr" grammar entry): it is mandatory to use things like "expression-1" or "simple expression" in the grammar.

In revised syntax, this case never occurs: when a rule needs an expression, it always uses the top level of the "expr" entry. The grammar is then simpler and easier to read and understand.

The choice of `"do"` followed by braces has something arbitrary. However, the keyword `"do"` let us easily think of something imperative (not functional). And the braces remind the sequence in the C language.

Why not `do..done`? Question of taste. It could have been `do..done`. The idea is to remain relatively discrete. And the proposed construction saves a keyword.

Note that a `let...in` in the sequence applies up to the end of the sequence, like in normal syntax. However, in normal syntax, because of the fact that the sequence is an opened construction, you can obtain strange results. In the example:

```
if condition then
   a-simple-statement;
statement-2;
statement-3;
```

Let us suppose that you need to add a let binding for the "simple statement": if you just add it, this is what you see:

```
if condition then
   let v = expr in
   a-simple-statement;
statement-2;
statement-3;
```

But what you get is actually:

```
if condition then
   let v = expr in
   a-simple-statement;
   statement-2;
   statement-3;
```

The `let` has "absorbed" the rest of the sequence, which is now included in the if condition. To be correct, you need to add an enclosing `begin..end` or parentheses.

5.4 Tuples and lists

- Parentheses are mandatory in tuples:

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 always 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.

Motivation to close the tuples by parentheses

In mathematics, tuples are always between parentheses.

Moreover, it is in a general policy of the revised syntax: close more constructions: it is easier to read and don't need to learn certain subtle precedences levels.

Motivation for the syntax of lists

In revised syntax, the lists are always closed. Be a "cons" `[a :: b]` or an enumeration of all items `[a; b; c]`, we always know syntactically where a list starts and when it ends.

This syntax have something similar of the lists in Lisp: the brackets are like the parentheses, the semicolons are like the spaces and the double colon is like the dot.

Moreover, the syntax:

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

is more understandable and more logical than the equivalent in normal syntax:

```
x :: y :: z :: t
```

Indeed, reading it in normal syntax, the types are not clear: `x`, `y` and `z` are not of same type than `t`, we have to remember that this double colon is right associative, which is generally not natural. In revised syntax, `x`, `y`, and `z` are at the same level (separated by semicolons), different from the one of `t` (separated from the rest by the double colon).

In revised syntax, it is clear that `x`, `y` and `z` are the first items of the list, because the syntax is identical when the list is ended by a "cons" and when it is not, what is not the case in normal syntax.

5.5 Irrefutable patterns

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

- A variable.
- The wildcard "`_`".
- The constructor "`()`".
- A tuple 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 definition, except "`()`", are not said "irrefutable" (the fact that they be alone or not cannot be determined at parsing time).

5.6 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 by brackets: an open bracket "`[`" before the first case, and a close bracket "`]`" after the last case:

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 mandatory. These examples work identically in normal and revised syntaxes:

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

Notice that in revised syntax, both `fun [ x -> x ]` and `fun x -> x` are correct. The currified pattern matching can be done with "`fun`" without brackets, 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 syntax:

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

- It is possible to use a construction "`where`", it is a reversed "`let`", but one can write only one bind:

```
 e where x = y
```

Motivation for one alone keyword "fun"

The presence of `fun` and `function` is somewhat strange, since they have the same semantics.

In revised syntax, by adding this notion of "irrefutable patterns", there is no ambiguity: a list not being an irrefutable pattern, the construction with brackets is not a parsing problem. When using an irrefutable pattern, there must be only one case, and therefore no close construction is necessary, allowing us to keep the simple frequent form: `fun x -> x`.

Motivation to close the constructions

It is to avoid the problem of the "dangling bar" (the same than the "dangling else" in the "if" construct). In normal syntax, this program:

```
match ... with
  case1 ->
    match ... with
      case11 -> ...
    | case12 -> ...
| case2 -> ...
```

is wrongly interpreted: to obtain what you want, you need to use parentheses or `begin..end` to close the internal `match` construct. There is a same problem with the `if` construct, because of the optional `else` (see further).

I admit that the fact that all cases do not start with the same token (the first starting with a left brace, the other ones with a vertical bar) is not practical in editing programs: it is indeed complicated to exchange the first case and the other ones. However readability and absence of ambiguity are more important than easiness to use and absence of verbosity: when it is easy to edit but risk to introduce bugs or irregularities, it is not sure that it be better.

Why not close the construction by a keyword, `end` for example, like the `Ada` language does? It is because an ending keyword gives an idea of something imperative, it does not make think that something is returned, which is however the case in the `match` construct, like most of `OCaml` constructs.

Motivation for the empty forms

The empty function is useful for initial cases of iterations or initial references values. It is not absolutely essential since it is possible to write:

```
fun _ -> assert False
```

The empty `match` existed before the introduction of the `assert` construction in `OCaml`. Like the assert, it indicates the position of the error in the file.

These constructions are there because they are the limit when the number of the matching cases reach zero.

Motivation for irrefutable patterns in "let"

In normal syntax, if you use a "let" binding with a non irrefutable pattern, you get a typing message "pattern matching is not exhaustive". If you want to be clean and add the missing cases, you have to torture your sources. Indeed, for example, the

```
let x :: y = a in b
```

must be changed into:

```
match a with x :: y -> b | ...
```

In revised syntax, since it is forbidden, you are never in this situation.

Motivation for the "where" construct

This construction existed in the old "Caml" V3.1 (whose development was stopped by the beginning of the 90ies) and I liked it much. There was a problem in this construct, because it was possible to add several bindings separated with "and", which sometimes could enter in conflict (another "dangling" case) with a possible "and" in an enclosing "let":

```
let a =
   b where c = d
and e = f in ...
```

In this situation, the "where" construct used to "absorb" the "and" of the "let" binding. The program was interpreted as:

```
let a =
   b where c = d and e = f
in ...
```

Because of that, in `Caml Light` and `OCaml`, the "where" construction were removed. But a "where" with only one binding could works. Anyway, having several bindings is not interesting nor useful nor readable, in this construction.

I personally use this construction in the case when the "let" binding is a function definition and the expression a call to this function. I generally prefer to write:

```
loop 0 where rec loop i = ...
```

than the equivalent form:

```
let loop i = ... in loop 0
```

I consider the form with `where` more readable in this situation.

5.7 Mutables and assignment

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

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

- The "`ref`" type is used as if its field label was 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`

Motivation

Having two constructions for the assignment is abnormal. In normal syntax, the `":="`, specific to the `ref` type, is an old rest of the time when references where implemented with a constructor (there were mutable constructors, then), and the codes to extract a reference value and to change it were complicated:

```
match x with Ref x -> x
match x with Ref x -> x <- y
```

It was then justified to have specific constructions `"!x"` and `"x := y"` for these cases. Now, references are implemented with a record type, and these constructions can be written:

```
x.contents
x.contents <- y
```

In normal syntax, there are 2 ways to access and assign references, although the method using the label "contents" is rarely used. In revised syntax, it is the only method. However, I consider "contents" as a too long identifier, it is why I changed it into "val". It is actually not a change in the definition of `ref` (since `Camlp4` does only syntax), it is changed in the syntax trees, the real name of the field remaining "contents".

As `":="` is no more necessary with the semantics of assigning a reference value, it can be used in the place of `"<-"`, a token less natural and introducing confusions (when we read it) with the `"->"` of the functions and pattern matchings.

The construction `!x` is no more necessary either since we can write `x.val`. We then save two tokens which were used only for the reference type.

5.8 Types

- The type constructors are before their type parameters, which are written with the currified form:

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 mandatory in tuples of types:

OCaml	Revised
`int * bool`	`(int * bool)`

- In declaration of a concrete type, brackets must enclose the constructors 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 tuple.

  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 tuple is done by using a tuple 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};`

Motivation for the applying order of type constructors

The order is to look like the constructors values: you can then read value in the same order than their types. The syntax with currification style is used also for value constructors.

Motivation for the abstract types syntax

It was to look like existential types, because abstract types are actually some kind of existential types. This may have a meaning if existential types are included one day in `OCaml`.

Motivation for the parentheses around tuple types

Close more constructions. Closed like tuples are. Moreover it is more visible in constructor declarations to differentiate the case of two parameters and one parameter being a tuple.

Motivation for the constructor declaration type

The revised syntax have tried to be the most general possible, to plan the possible future extensions of the language.

Record types are closed by braces (no change). Symmetrically, the sum types (declaring constructors) are closed by brackets. This is also a way to consider them just as "types". We could imagine that they be authorized one day outside type declarations. For example like this:

```
fun (x : [ A | B ]) -> ...
type t = { lab : [ A | B ] }
type u = [ C of { lab : ...} ]
```

The form of the last line is, by the way, the method used in the language `SML`, where record types are always anonymous.

In `Camlp4` abstract syntax, there is no notion of "type declaration": a type declaration is just a type. The fact that sum types and record types are accepted only in type declarations is done when converting into the abstract syntax which `ocamlc` uses.

Motivation for the empty type

As the type constructor definition is closed, it is possible to imagine the empty type. Not very useful, but we have it without any cost: a type inhabited by nothing (empty set).

Motivation for the currified syntax for constructors

This reflects the actual semantics. There are indeed two cases, and the values in the two cases are implemented differently. The arity of constructors are more clear.

In normal syntax, it is difficult to understand (and to explain) why if C is a constructor with two parameters, this is accepted:

```
fun C (x, y) -> (x, y)
```

but not that:

```
fun C x -> x
```

In revised syntax you have to write:

```
fun [ C x y -> (x, y) ]
```

The revised syntax reflects the fact that the two parameters of the constructor `C` cannot be considered as a tuple.

This does not mean that the "partial evaluation" of constructors is accepted: accept it or not is a semantic issue, treated at `OCaml` typing time.

Motivation for the uppercase for True and False

In normal syntax, `true` and `false` are the only constructors which start with a lowercase letter. It is due to historical reasons: in `Caml Light`, no constructors (of any type) need to be capitalized. When `OCaml` was created, this was changed, but strangely, `true` and `false` escaped to this rule. They are even now considered as keywords, what they should not be, since they are not syntactic constructs or part of syntactic constructs.

In revised syntax, they must be written `True` and `False` and are not keywords.

Motivation for mutable syntax in records

It is just to read: "the label x is a mutable integer" instead of "the mutable label x is an integer", which is less clear.

5.9 Modules

Modules application uses the currified form:

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

Motivation

Currification syntax is more natural in functional languages. There is no reason to have two different syntaxes for applications (whatever we apply): one with parentheses, one with currification.

5.10 Classes and objects

The classes and 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
```

(documentation to be updated)

5.11 Miscellaneous

- The "else" is mandatory in the "if" statement:

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

- The boolean operations "or" and "and" must be written only 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 when needed.
- 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.
- 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;
```

Motivation for the "else"

The else is mandatory to avoid the "dangling else" problem. In normal syntax, you can write:

```
if a then
  if b then c
else d
```

In the above program, the "else d" will actually corresponds to the "if b" not to the "if a". In revised syntax, the "else" being mandatory, the problem does not exist.

`OCaml` being a functional language, it is normal that the "else" case be mandatory: indeed if the condition is false, what is returned by the statement is not clear in normal syntax.

All these "dangling" problems cause also problems in pretty printing: it is not easy to know if the constructions have to be parenthesized or not. In revised syntax, there are no dangling problems and no problem in pretty printing. To pretty print in normal syntax, a solution had to be used, using an extra parameter transmitted in all functions.

We remark that in revised syntax, the `if` construct is not closed, it does not need to be.

Motivation for the "or" and "and" operators

There is no reason to accept two syntaxes for the "or" operator and two for the "and" operator. The syntaxes `or` and `&` are actually old constructions, kept for an old backward compatibility.

Motivation for the suppression of begin..end

In normal syntax, the construction with `begin` and `end` is actually the same than the parentheses: often a question of personal taste. In normal syntax, when parenthesis is necessary, some programmers prefer `"begin match...end"`, other `"(match...)"`.

In revised syntax, the cases when such a parenthesization is necessary is much less frequent, since most constructions are already parenthesized. Two constructions for that are not necessary.

Motivation for syntax for alone operators

To avoid the case of the `*` operator which must be specifically written with spaces around it, since `(*)` in lexically interpreted as a beginning of a comment.

Motivation for the fact that there are no automatic infixes

Since we are under Camlp4, we can use Camlp4 features.

Motivation for the "declare" construction

Essential when a syntax extension in `OCaml` structure item generates several structure items. For example, if you make a syntax change in order that a type declaration generates 1/ the type declaration itself and 2/ functions to be applied to this type.

When converted into `OCaml` normal syntax tree, this construct is inlined.

5.12 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:

<table>
<tr><td>OCaml</td><td>Revised</td></tr>
<tr><td><pre>parser
  [< 'Foo >] -> e
| [< p = f >] -> f</pre></td><td><pre>parser
[[: `Foo :] -> e
| [: p = f :] -> f]</pre></td></tr>
<tr><td><code>parser [< 'x >] -> x</code></td><td><code>parser [[: `x :] -> x]</code></td></tr>
<tr><td><code>parser [< 'x >] -> x</code></td><td><code>parser [: `x :] -> x</code></td></tr>
</table>

- 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 []
```

Motivation for the keyword "parser", rather than "parse"

Actually, it is not different from the choice of the normal syntax, since the same keyword is used.

The keyword "parser" is like "function", not like "match". The "match" and "try" statements are direct actions, with their immediate parameters. On the other hand, the parsers and functions are just "concepts": they are not immediately applied with their parameters. One must read: "this is a parser" just like "this is a function".

The word "parse" might have been used if the construction was "parse xxx with". This is written "match xxx with parser" in order to save a keyword.

Motivation for `[:` instead of `[<`

It is a question of readability, because of the presence of quotations in our extended language, whose syntax use many "less" and "greater" characters. And it is a problem for a list of quoted things:

```
[<:expr< xx >>; <:expr< yy >>]
```

Motivation for quotes and backquotes

Actually, this should have been done in `OCaml` normal syntax, since from `Caml Light` to `OCaml`, the character used to enclose characters changed from backquote into right quote. It would have been then normal to invert that for the streams terminals, but it was forgotten.

In normal syntax, this creates sometimes problems in characters streams:

```
parser [< '('a' | 'b') >] -> ...
```

The lexer interprets the first parenthesis as a character, which causes thus parsing error. You must add a space before the left parenthesis:

```
parser [< ' ('a' | 'b') >] -> ...
```

In revised syntax, which backquotes, this problem does not appear.

Motivation for closing the syntax of parsers

To resolve the same problem of "dangling bar" than for functions, matches and tries. This syntax is closed the same way.

Motivation for the empty parser

Useful in initial cases in iterations or initial references values.

Chapter 6

OCaml syntax trees

This chapter explains how to create OCaml syntax trees using quotations.

The OCaml syntax tree is defined in the module MLast (file "mLast.mli"): it is of course always possible to create your syntax tree nodes by combining values of these types. But, for that, you need a precise documentation of all the constructors, how they have to be used, and which syntactic construct they correspond to.

Chapter 4 explained us that quotations are helpful to represent abstract values with concrete syntax. This is very adapted for abstract syntax trees, since they have corresponding concrete syntaxes. In this case, you have nothing to learn: if you know the concrete syntax, you know the abstract syntax. Actually, you need to know the revised syntax (chapter 5) because it is the syntax used in these quotations.

The provided quotation extension is the file q_MLast.cmo. Once this file loaded, you can write any complicated piece of program in concrete syntax, e.g.:

```
<:expr< match f $x$ with [ $uid:p$ -> $x$ | C z -> 7 ] >>
```

This example corresponds exactly to this code (written in normal syntax):

```
MLast.ExMat
 (loc, MLast.ExApp (loc, MLast.ExLid (loc, "f"), x),
  [MLast.PaUid (loc, p), None, x;
   MLast.PaApp
    (loc, MLast.PaUid (loc, "C"), MLast.PaLid (loc, "z")), None,
   MLast.ExInt (loc, "7")]);;
```

These two forms are absolutely equivalent, but the first one is much simpler and more readable. Notice the expressions between dollar signs `$`: they are antiquotations (see chapter 4). We are going to see how to use them.

For the moment, here is the list of the quotations defined in the file q_MLast.cmo:

- `expr` to create expression nodes
- `patt` to create pattern nodes

- `ctyp` to create type nodes
- `sig_item` to create signature item nodes
- `str_item` to create structure item nodes
- `module_type` to create module type nodes
- `module_expr` to create module expression nodes

Each of these quotations use the revised syntax (chapter 5). You have to know that their expression form (i.e. not pattern) all contain the variable named `"loc"`. It allows you to specify a source location to the syntax tree. There must then exist a variable `"loc"` visible in the environment where the quotation is used.

Notice that in the `Camlp4` grammar system (chapter 3), the variable `"loc"` in defined in all semantic actions of the rules: Camlp4 grammar system and Camlp4 quotations of abstract syntax trees are made to work together.

6.1 Antiquotations

The antiquotations allow to insert abstract syntax trees inside other abstract syntax trees. They are enclosed by dollar signs `$`. Between these dollars, you can put any expression or pattern in the enclosing syntax.

For example, if you have a variable `"x"` of type `MLast.expr`, you may want to create "the tree node corresponding to the call of some function `"f"` with this expression. This can be written:

```
<:expr< f $x$ >>
```

Note that if the quotation is is position of pattern, the content of the quotation must be a pattern: here are legal pieces of program (in normal syntax):

```
function <:expr< f $x$ >> -> x;;
function <:expr< f $_$ >> -> 1;;
```

6.2 Antiquotations labels

Some of these antiquotations may be used with a label (an identifier ending with a colon character `":"` in the beginning of the antiquotation). It is for the case when the concrete syntax is not enough to know what we are talking about.

For example, if `x` is a variable of type string, we need ways to create 1/ the tree node corresponding to the variable whose name is the value of `x` 2/ the tree node corresponding to the string whose content is the value of `x` 3/ the tree node corresponding to the constructor whose name is the value of `x`.

Labels allow to do that: in the first case, it must be `"lid"` (for "l(owercase) id(entifier)"), in the second case, `"str"` (for "str(ing)"), in the third case, `"uid"` (for "u(ppercase) id(entifier)"), i.e. respectively:

```
<:expr< f $lid:x$ >>
<:expr< f $str:x$ >>
<:expr< f $uid:x$ >>
```

Therefore:

```
let x = "foo" in <:expr< f $lid:x$ >> is equivalent to: <:expr< f foo >>
let x = "foo" in <:expr< f $str:x$ >>       "           <:expr< f "foo" >>
let x = "Foo" in <:expr< f $uid:x$ >>       "           <:expr< f Foo >>
```

Important remark: the antiquotations of type string (str) and character (chr) must contain an "escaped" string: some characters like backslashes, quotes, doubles quotes must be prefixed by a backslash. To be sure, use the OCaml library functions `"String.Escaped"` and `"Char.Escaped"`.

A label often encountered is `"list"`: it is when the syntactic construct may expect a list at this place. Here are examples with two antiquotations, one being labelled, the other not:

```
<:expr< [| $list:x$ |] >>
<:expr< [| $x$ |] >>
```

- In the first case, `x` must be a value of type `MLast.expr` list, and the quotation represents an array whose elements are in `x`.
- In the second case, `x` if of type `MLast.expr`, and the quotation represents an array of just one element.

Notice that the antiquotations must respect the enclosing syntax rules: if `x` and `y` are of type `MLast.expr` and `z` of type `MLast.expr` list, the code:

```
<:expr< [| $list: x::y::z $ |] >>
```

is correct in normal syntax, but if you use the revised syntax, you must write it:

```
<:expr< [| $list: [x; y :: z] $ |] >>
```

6.3 Description

Well, we could describe all nodes one by one, but they are actually already described in the reference manual. Refer to it.

Chapter 7

Extending the syntax of OCaml

7.1 Introduction

Syntax extensions in `OCaml` can be done by extending the grammars entries of the OCaml syntax. All grammars entries are defined in the module named `Pcaml`. They all return values of types defined in the module `MLast`: nodes of these types can be created using the predefined quotation expansion `q_MLast.cmo`.

The entries in `Pcaml` are:

- `expr` for expressions, returning values of type `MLast.expr`.
- `patt` for patterns, returning values of type `MLast.patt`.
- `ctyp` for types, returning values of type `MLast.ctyp`.
- `module_type` for module types, returning values of type `MLast.module_type`.
- `module_expr` for module expressions, returning values of type `MLast.module_expr`.
- `sig_item` for signature items, returning values of type `MLast.sig_item`.
- `str_item` for structure items, returning values of type `MLast.str_item`.

Most of these entries are generally defined ("extended") with several "levels" (see chapter 3). Some of them are labelled, in order to be able to extend them or to insert other levels.

The levels and their possible labels are not predefined. It depend on how the syntax define them. To see which labels are defined and which rule they contain, enter the toplevel and type for the normal syntax:

```
#load "camlp4o.cma";;
Grammar.Entry.print Pcaml.expr;; (* for the expressions *)
Grammar.Entry.print Pcaml.patt;; (* for the patterns *)
                                 (* ... and so on *)
```

For the revised syntax, load "camlp4r.cma" instead. If you defined another syntax of the whole language or want to see the other syntaxes provided, load it before, and call Grammar.Entry.print of the desired grammar entry. Look at the manual page (man camlp4 in the shell) to see all available syntaxes and extensions.

Once you have the list of the grammar entry you want to extend and the possible level label, you can do your extension.

7.2 Example: "repeat until" like in Pascal

If you read all this tutorial, you are able to understand this complete example. If you did not, just create the files and type the indicated commands.

Write first a file named foo.ml containing:

```
open Pcaml;;
EXTEND
  expr: LEVEL "expr1"
    [[ "repeat"; e1 = expr; "until"; e2 = expr ->
          <:expr< do { $e1$; while not $e2$ do { $e1$; } } >> ]];
END;;
```

The compilation of this file can be done by typing under the shell (the dollar is the shell prompt):

```
$ ocamlc -pp "camlp4o pa_extend.cmo q_MLast.cmo" -I +camlp4 \
    -c foo.ml
```

Here is the file bar.ml containing a repeat..until statement:

```
let main () =
  let i = ref 0 in
  repeat print_int !i; incr i until !i = 10;
  print_newline ()
let _ = main ()
```

You can compile it by typing:

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

And run it:

```
$ ./a.out
0123456789
```

Or just pretty print the program with the expanded syntax:

```
$ camlp4o ./foo.cmo pr_o.cmo bar.ml
let main () =
  let i = ref 0 in
  begin
```

```
    begin print_int !i; incr i end;
    while not (!i = 10) do print_int !i; incr i done;
  end;
  print_newline ()
;;
main ();;
```

7.3 Example: a constant

If you want to have the equivalent of a `#define` of C, you can write for example, if you want `FOO` to be replaced by `54` in expressions and patterns:

```
open Pcaml;;
EXTEND
  expr: LEVEL "simple"
    [[ UIDENT "FOO" -> <:expr< 54 >> ]];
  patt: LEVEL "simple"
    [[ UIDENT "FOO" -> <:patt< 54 >> ]];
END;;
```

The compilation of this file can be done by typing:

```
$ ocamlc -pp "camlp4o pa_extend.cmo q_MLast.cmo" -I +camlp4 \
    -c foo.ml
```

Here is the file `bar.ml` containing `FOO` constants:

```
FOO;;
function FOO -> 22;;
```

You can compile it by typing:

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

You can just pretty print the program with the expanded syntax:

```
$ camlp4o ./foo.cmo pr_o.cmo bar.ml
54;;
function 54 -> 22;;
```

7.4 Example: a "for" loop like in C

Here is an example of an syntax extension allowing to write a "for" loop like in C. A construction is added with the loop variable and 3 parameters, simple expressions: the first one is the initial value, the second the test, the third the way to change the loop variable.

Note that we use here the directives `#load` inside the source of the syntax extension, allowing to parse it with `camlp4o` without having to specify these files in the command line.

File "cloop.ml":

```
#load "q_MLast.cmo";;
#load "pa_extend.cmo";;

open Pcaml

let gensym =
  let cnt = ref 0 in
  fun var ->
    let x = incr cnt; !cnt in
    var ^ "_gensym" ^ string_of_int x

let gen_for loc v iv wh nx e =
  let loop_fun = gensym "iter" in
  <:expr<
    let rec $lid:loop_fun$ $lid:v$ =
      if $wh$ then do { $e$; $lid:loop_fun$ $nx$ } else ()
    in
    $lid:loop_fun$ $iv$ >>

EXTEND
  expr: LEVEL "expr1"
    [ [ "for"; v = LIDENT; iv = expr LEVEL "simple";
        wh = expr LEVEL "simple"; nx = expr LEVEL "simple";
        "do"; e = expr; "done" ->
          gen_for loc v iv wh nx e ] ]
  ;
END
```

Compile this file with:

```
$ ocamlc -pp camlp4o -I +camlp4 -c cloop.ml
```

Example under the toplevel:

```
$ ocaml
        Objective Caml version 3.02+7 (2001-09-29)

# #load "camlp4o.cma";;
        Camlp4 Parsing version 3.02+7 (2001-09-29)

# #load "cloop.cmo";;
# for i = 0 to 10 do print_int i; done;; (* normal loop *)
012345678910- : unit = ()
# for c 0 (c<10) (c+1) do print_int c; done;;
0123456789- : unit = ()
# for c 0 (c<10) (c+3) do print_int c; done;;
0369- : unit = ()
```

Exemple of compilation of a program using this construction:

```
$ cat foo.ml
for c 0 (c<10) (c+2) do print_int c; done
$ ocamlc -pp "camlp4o ./cloop.cmo" -c foo.ml
```

And if you want to see the generated program (for example to check that the extension is correct):

```
$ camlp4o ./cloop.cmo pr_o.cmo foo.ml
let rec iter_gensym1 c =
  if c < 10 then begin print_int c; iter_gensym1 (c + 2) end
in
iter_gensym1 0;;
```

7.5 Example: generating printers of types

We are going to define a syntax extension, so that for all types definitions, the definition of printers of the values of this types is automatically added. In this example, we limit to sum types (types with constructors), but it can be easily extensible for record types, abstract types, types renaming.

7.5.1 First version: monomorphic sum types with constant constructors

The example, which is going to be our test, is the following file "`col.ml`":

```
type colour = Red | Green | Blue
```

We want that, when preprocessed with the correct syntax extension, this file be interpreted like this:

```
type colour = Red | Green | Blue
let print_colour =
  function
    Red -> print_string "Red"
  | Green -> print_string "Green"
  | Blue -> print_string "Blue"
```

The syntax extension will be defined in the following file "`pa_type.ml`". As a beginning, let us just see how we insert the grammar rule. The function "`gen_print_funs`" generating the printer functions just generates a phony statement:

```
#load "pa_extend.cmo";;
#load "q_MLast.cmo";;

let gen_print_funs loc tdl =
  <:str_item< not yet implemented >>

let _ =
  EXTEND
    Pcaml.str_item:
      [ [ "type"; tdl = LIST1 Pcaml.type_declaration SEP "and" ->
```

```
                let si1 = <:str_item< type $list:tdl$ >> in
                let si2 = gen_print_funs loc tdl in
                <:str_item< declare $si1$; $si2$; end >> ] ]
      ;
    END
```

Remark the "declare" statement in the "str_item" syntax tree at the end of this file, destinated to group two structure items together: 1/ the type definition 2/ the printer.

This file can be compiled like this:

```
$ ocamlc -pp camlp4o -I +camlp4 -c pa_type.ml
```

We can test the example file, "col.ml", but not yet with the compiler, since it would generate semantic error because of the "not yet implemented" statement. Let us test it therefore with a pretty printer:

```
$ camlp4o ./pa_type.cmo pr_o.cmo col.ml
<W> Grammar extension: in [str_item], some rule has been masked
type colour =
    Red
  | Green
  | Blue
let _ = not yet implemented
```

See the extra generated statement "not yet implemented". You remark, also, that there is a warning in the beginning: it means that the syntax rule we added in str_item was already present in the grammar we used.

To avoid such a warning message, the solution is to add, before the EXTEND statement, a DELETE_RULE statement:

```
DELETE_RULE
  Pcaml.str_item: "type"; LIST1 Pcaml.type_declaration SEP "and"
END;
```

Let us attack now the function gen_print_funcs. It receives a list (since the "type" declaration can define several types, possibly mutually recursive) of types definitions. We know that we have to generate a definition, recursive, with as many printing functions as types. The following function, gen_one_type_print_fun, will generate the printer for one type definition. For the moment, the body is a "not yet implemented" statement:

```
let fun_name n = "print_" ^ n

let gen_one_print_fun loc ((loc, n), tpl, tk, cl) =
  <:patt< $lid:fun_name n$ >>, <:expr< not yet implemented >>

let gen_print_funs loc tdl =
  let pel = List.map (gen_one_print_fun loc) tdl in
  <:str_item< value rec $list:pel$ >>
```

Recompile the syntax expander file with these functions and test with "`col.ml`": you can see a function named "`print_colour`".

Let use improve now "`gen_one_print_fun`". It has to generate a let binding definition, composed of the couple of a pattern (the name of the function) and an expression. Our function receives as parameter a type definition which is a t-uple of 4 values: 1/ the type name (with his location), 2/ the list of its possible parameters, 3/ the type kind (a type, actually) and 4/ a list of possible constraints.

In a first version, we are going to ignore the type parameters "`tpl`": we see later how they intervene in the generated function and our code will work, for the moment, only for monomorphic types.

We limit also to the "sum" types (i.e. types with constructors); for other types kinds, we shall generate a function which fails.

We added the function "`gen_print_sum`" which treats a sum type by generating a match association for each constructor (function "`gen_print_cons`") and building the function with the resulting list.

That function "`gen_print_cons`" gets a constructor definition, i.e. a tuple with: 1/ a location, 2/ a string (the constructor name) and 3/ a list of types parameters (ctyp list). We ignore for the moments the constructors parameters. The function "`gen_print_cons_patt`" generates the pattern part of the case, and "`gen_print_cons_expr`" the expression part of the function, the print statement:

Here is a first (but complete) version of our syntax extension (file "`pa_type.ml`"):

```
#load "pa_extend.cmo";;
#load "q_MLast.cmo";;

let fun_name n = "print_" ^ n

let gen_print_cons_patt loc c tl =
  <:patt< $uid:c$ >>

let gen_print_cons_expr loc c tl =
  <:expr< print_string $str:c$ >>

let gen_print_cons (loc, c, tl) =
  let p = gen_print_cons_patt loc c tl in
  let e = gen_print_cons_expr loc c tl in
  p, None, e

let gen_print_sum loc cdl =
  let pwel = List.map gen_print_cons cdl in
  <:expr< fun [ $list:pwel$ ] >>

let gen_one_print_fun loc ((loc, n), tpl, tk, cl) =
```

```
  let body =
    match tk with
      <:ctyp< [ $list:cdl$ ] >> -> gen_print_sum loc cdl
    | _ -> <:expr< fun _ -> failwith $str:fun_name n$ >>
  in
  <:patt< $lid:fun_name n$ >>, body

let gen_print_funs loc tdl =
  let pel = List.map (gen_one_print_fun loc) tdl in
  <:str_item< value rec $list:pel$ >>

let _ =
  DELETE_RULE
    Pcaml.str_item: "type"; LIST1 Pcaml.type_declaration SEP "and"
  END;
  EXTEND
    Pcaml.str_item:
      [ [ "type"; tdl = LIST1 Pcaml.type_declaration SEP "and" ->
            let si1 = <:str_item< type $list:tdl$ >> in
            let si2 = gen_print_funs loc tdl in
            <:str_item< declare $si1$; $si2$; end >> ] ]
    ;
  END
```

We can recompile this version, and test on the example file "`col.ml`" by pretty printing the result:

```
$ ocamlc -pp camlp4o -I +camlp4 -c pa_type.ml
$ camlp4o ./pa_type.cmo pr_o.cmo col.ml
type colour =
    Red
  | Green
  | Blue
let rec print_colour =
  function
    Red -> print_string "Red"
  | Green -> print_string "Green"
  | Blue -> print_string "Blue"
```

It is what we wanted! This can be used, now, directly with the compiler without the pretty printing phase:

```
$ ocamlc -pp "camlp4o ./pa_type.cmo" -c col.ml
```

We could also add the directive "`#load "./pa_type.cmo";;`" in the beginning of "`col.ml`" and just compile with:

```
$ ocamlc -pp camlp4o -c col.ml
```

but it is not a good idea, since we may want to use the same source with the preprocessing or without it.

7.5.2 Second version: constructors with parameters

Let us add the case of sum types having constructors with parameters. Our example for testing that will be the definition of lambda terms of section 4.7. File "`term.ml`":

```
type term =
    Var of string
  | Func of string * term
  | Appl of term * term
```

The desired result should be something like this:

```
type term =
    Var of string
  | Func of string * term
  | Appl of term * term
let rec print_term =
  function
    Var x1 ->
      print_string "Var"; print_string " ("; print_string x1;
      print_string ")"
  | Func (x1, x2) ->
      print_string "Func"; print_string " ("; print_string x1
      print_string ", "; print_term x2; print_string ")"
  | Appl (x1, x2) ->
      print_string "Appl"; print_string " ("; print_term x1
      print_string ", "; print_term x2; print_string ")"
```

Like in the above desired result, we decide to name the parameters with "`x`" followed by the number of the parameter, defined by the following function "`param_name`":

```
let param_name cnt = "x" ^ string_of_int cnt
```

We need a function "`list_mapi`", which is like "`List.map`" but the function applied receives the number of the list element as first parameter. This allows us to generate the name of the constructor parameter while exploring the type list:

```
let list_mapi f l =
  let rec loop cnt =
    function
      x :: l -> f cnt x :: loop (cnt + 1) l
    | [] -> []
  in
  loop 1 l
```

The function "`gen_print_cons_patt`" which treats the pattern part of the match association, is changed like this:

```
let gen_print_cons_patt loc c tl =
  let pl =
    list_mapi (fun n _ -> <:patt< $lid:param_name n$ >>)
      tl
  in
  List.fold_left (fun p1 p2 -> <:patt< $p1$ $p2$ >>)
    <:patt< $uid:c$ >> pl
```

With these changes, the pattern part of the generated function "`print_term`" is correct. Test it.

For the expression part, we have to generate the call to the printers for all the constructors parameters. We add a function "`gen_print_type`" to generate a printer associated with a type. For the moment, it just generates it for a simple type name. For other types, it generates a printer displaying an ellipsis:

```
let gen_print_type loc =
  function
    <:ctyp< $lid:s$ >> -> <:expr< $lid:fun_name s$ >>
  | _ -> <:expr< fun _ -> print_string "..." >>
```

We need also a function which generates the call to this printer function with the constructor parameter:

```
let gen_call loc n f = <:expr< $f$ $lid:param_name n$ >>
```

and a function adding the extra syntax: spaces, parentheses and commas:

```
let gen_print_con_extra_syntax loc el =
  let rec loop =
    function
      [] | [_] as e -> e
    | e :: el -> e :: <:expr< print_string ", " >> :: loop el
  in
  <:expr< print_string " (" >> :: loop el @
  [<:expr< print_string ")" >>]
```

Now, we can change the function "`gen_print_cons_expr`" using all these functions:

```
let gen_print_cons_expr loc c tl =
  let pr_con = <:expr< print_string $str:c$ >> in
  match tl with
    [] -> pr_con
  | _ ->
      let pr_params =
        let type_funs = List.map (gen_print_type loc) tl in
        list_mapi (gen_call loc) type_funs
      in
      let pr_all = gen_print_con_extra_syntax loc pr_params in
      let el = pr_con :: pr_all in
      <:expr< do { $list:el$ } >>
```

Grouping all these functions together, you can make a second version of "pa_type.ml" which works with the file "term.ml". Test it! Try it also with your own programs having sum type definitions.

7.5.3 Third version: polymorphic types

This time, we are going to generate the good code for polymorphic types, i.e. types defined with types variables. Our example will be the definition of the type "mlist" like this. File "mlist.ml":

```
type 'a mlist = Nil | Cons of 'a * 'a mlist
```

The printer of such a type will receive as parameter the print functions of the instantiated type. As many as the type has type variables. We can then call "print_mlist print_int" for an "int mlist", "print_mlist print_string" for a "string mlist" and so on.

The desired result for the type "mlist" is:

```
type 'a mlist = Nil | Cons of 'a * 'a mlist
let rec print_mlist pr_a =
  function
    Nil -> print_string "Nil"
  | Cons (x1, x2) ->
      print_string "Cons"; print_string " ("; pr_a x1;
      print_string ", "; print_mlist pr_a x2; print_string ")"
```

The name of the printer function for a type variable will be "pr_" followed by the type variable name:

```
let fun_param_name n = "pr_" ^ n
```

To add the function parameters to the printer definition ("let print_mlist pr_a = ..." in our example), we change our function "gen_one_print_func" by inserting them in the body of the function, just before its result, like this:

```
  let body =
    List.fold_right
      (fun (v, _) e ->
         <:expr< fun $lid:fun_param_name v$ -> $e$ >>)
      tpl body
  in
```

For the printing of a type variable ("pr_a x1" in our example), we add the case of type variables in our function "gen_print_type":

```
  | <:ctyp< '$s$ >> -> <:expr< $lid:fun_param_name s$ >>
```

And to generate the printing of types with parameters (we have a recursive case in our example: "print_mlist pr_a x2" for the constructor parameter of type "'a mlist"), we add, in the same function, the case of types applications. But since it needs a recursive call, the function "gen_print_type" is rewritten with a internal recursive definition.

Here is the complete version:

```
#load "pa_extend.cmo";;
#load "q_MLast.cmo";;

let fun_name n = "print_" ^ n
let fun_param_name n = "pr_" ^ n
let param_name cnt = "x" ^ string_of_int cnt

let list_mapi f l =
  let rec loop cnt =
    function
      x :: l -> f cnt x :: loop (cnt + 1) l
    | [] -> []
  in
  loop 1 l

let gen_print_type loc t =
  let rec eot =
    function
      <:ctyp< $t1$ $t2$ >> -> <:expr< $eot t1$ $eot t2$ >>
    | <:ctyp< $lid:s$ >> -> <:expr< $lid:fun_name s$ >>
    | <:ctyp< '$s$ >> -> <:expr< $lid:fun_param_name s$ >>
    | _ -> <:expr< fun _ -> print_string "..." >>
  in
  eot t

let gen_call loc n f = <:expr< $f$ $lid:param_name n$ >>

let gen_print_cons_patt loc c tl =
  let pl =
    list_mapi (fun n _ -> <:patt< $lid:param_name n$ >>)
      tl
  in
  List.fold_left (fun p1 p2 -> <:patt< $p1$ $p2$ >>)
    <:patt< $uid:c$ >> pl

let gen_print_con_extra_syntax loc el =
  let rec loop =
    function
      [] | [_] as e -> e
    | e :: el -> e :: <:expr< print_string ", " >> :: loop el
  in
  <:expr< print_string " (" >> :: loop el @
  [<:expr< print_string ")" >>]

let gen_print_cons_expr loc c tl =
  let pr_con = <:expr< print_string $str:c$ >> in
  match tl with
```

```
    [] -> pr_con
  | _ ->
      let pr_params =
        let type_funs = List.map (gen_print_type loc) tl in
        list_mapi (gen_call loc) type_funs
      in
      let pr_all = gen_print_con_extra_syntax loc pr_params in
      let el = pr_con :: pr_all in
      <:expr< do { $list:el$ } >>

let gen_print_cons (loc, c, tl) =
  let p = gen_print_cons_patt loc c tl in
  let e = gen_print_cons_expr loc c tl in
  p, None, e

let gen_print_sum loc cdl =
  let pwel = List.map gen_print_cons cdl in
  <:expr< fun [ $list:pwel$ ] >>

let gen_one_print_fun loc ((loc, n), tpl, tk, cl) =
  let body =
    match tk with
      <:ctyp< [ $list:cdl$ ] >> -> gen_print_sum loc cdl
    | _ -> <:expr< fun _ -> failwith $str:fun_name n$ >>
  in
  let body =
    List.fold_right
      (fun (v, _) e ->
         <:expr< fun $lid:fun_param_name v$ -> $e$ >>)
      tpl body
  in
  <:patt< $lid:fun_name n$ >>, body

let gen_print_funs loc tdl =
  let pel = List.map (gen_one_print_fun loc) tdl in
  <:str_item< value rec $list:pel$ >>

let _ =
  DELETE_RULE
    Pcaml.str_item: "type"; LIST1 Pcaml.type_declaration SEP "and"
  END;
  EXTEND
    Pcaml.str_item:
      [ [ "type"; tdl = LIST1 Pcaml.type_declaration SEP "and" ->
            let si1 = <:str_item< type $list:tdl$ >> in
            let si2 = gen_print_funs loc tdl in
            <:str_item< declare $si1$; $si2$; end >> ] ]
```

```
      ;
    END
```

7.5.4 Improvements

It is possible to add, the same way, the other kind of types: record types, abstract types, and so on.

Another interesting improvement is to generate, instead of "`print_string`" statements, functions of the "`Format`" library, with pretty printing boxes.

Further, that version can be still improved, by generating only one "`Format.fprintf`" by printing case (instead of a sequence of printing statements), using the very useful abbreviations provided by that library by the prefixes "`@`" inside the format strings.

Chapter 8

The camlp4 commands

Camlp4 is a command which you can type from your shell. It has several versions: `camlp4`, `camlp4o`, `camlp4r`. Actually, the basic command is `camlp4`, the other ones being shortcuts for this command with syntaxes previously loaded (see man page).

Its options are in two parts: the "load" options and the "other" options. The input file is among the "other" options, generally a `.mli` interface file or `.ml` implementation file.

8.1 The load options

The "load" options *must* come first. They are a list of OCaml object files (cmo) or library file (cma) which are loaded in camlp4 core before any other operation. These files are found in a path. You can use the option `-I` to add directories in this path. By default, the path contains only the camlp4 library directory (depending on the installation).

Notice that the current directory is *not* by default in the path: to load a file belonging to the current directory, you have to use either the option `-I .` or to type the object file prefixed by a dot and a slash.

In fact, the command `camlp4` alone, without any load option does nothing with the input file and fails. You need to specify at least one parser and one printer. If you don't specify any parser, you get the message "Failure: no loaded parsing module", if you don't specify any printer, "Failure: no printer". When it runs, `camlp4` applies the parser to the input file, which builds a syntax tree and this syntax tree is printed using the printer.

8.2 The other options

The other options follow. If there is a risk of ambiguities, you can use `--` to separate the load options and the other ones: indeed, the other options can be extended by the object files loaded in the core: in particular, the provided object file `pr_depend.cmo` adds `-I` among the other options.

To see the list of available options, type `camlp4 -help` or, since the object files can extend the options, type `camlp4 <load-options> -help` where `<load-options>` is your load options. Type, for example, and compare the results:

```
camlp4 -help
camlp4 pa_r.cmo pa_extend.cmo -help
camlp4 pr_depend.cmo -help
```

8.3 The predefined object files

The predefined object files, available in the camlp4 library directory, are of two kinds:

- The parser files, which start with `"pa_"` for syntax definitions or extensions ("pa" like "PArser"), or with `"q_"` for quotations expanders.
- The printer files, which start with `"pr_"` ("pr" like "PRinter").

You can see the list of available parsers and printers in the camlp4 library directory or using the man page. The main one are:

- `pa_o.cmo`: the parser with normal OCaml syntax
- `pa_r.cmo`: the parser with revised syntax
- `pr_dump.cmo`: the printer of the raw abstract syntax tree
- `pr_o.cmo`: the printer with normal OCaml syntax
- `pr_r.cmo`: the printer with revised syntax

Note that `pr_dump.cmo` displays the abstract syntax tree in binary. By default, it is displayed in standard output (except if the option `-o` is used). Thus, pay attention: it is binary material, if you don't redirect the output to a file, you are likely to perturb your terminal, because it may find escape sequences and interpret them. This printer is used as input to the OCaml compiler.

The other printers are useful to display the result of a parsing, test syntax extensions or quotations expanders. You can print either in normal syntax or in revised syntax. This can be used also to convert from normal to revised syntax and vice-versa.

Printers can also be extended. This is undocumented because the printing facility is not perfected and is a little bit complicated to use. But you can use the predefined printers. They rebuild the text in the initial form. For example `pr_extend.cmo` rebuilds the `EXTEND` statements.

An interesting predefined quotation expander is `q_phony.cmo` which convert all quotations into phony variables. The quotations are therefore printed as they are. Useful to pretty print your program containing quotations without expanding them.

8.4 Commands camlp4o and camlp4r

The commands `camlp4o` and `camlp4r` are actually shortcuts for (respectively):

```
camlp4 pa_o.cmo pa_op.cmo pr_dump.cmo
camlp4 pa_r.cmo pa_rp.cmo pr_dump.cmo
```

You can add parsers and (other) printers to these commands also.

8.5 Toplevel

To use Camlp4 in the toplevel, you have to first load the camlp4 machinery. Just loading `pa_r.cmo` or `pa_o.cmo` would not work: the toplevel need a specific object file which contains 1/ a plug-in to tell it to parse using the Camlp4 stuff 2/ the camlp4 library 3/ object files used by the camlp4 parsers (the camlp4 pretty printing does not work in the toplevel).

There are two files which contain all of this: `camlp4r.cma` starting with the revised syntax and `camlp4o.cma` starting with the normal syntax. You must then load one of them by typing either:

```
        #load "camlp4r.cma";;
or
        #load "camlp4o.cma";;
```

From now, you are in the Camlp4 machinery. Everything you type is parsed by Camlp4. In particular, if you chose the revised syntax, any other `#load` statement must end with an alone semicolon, as the revised syntax requires. To use the camlp4 grammar system, load `pa_extend.cmo`, to use the predefined quotations for the OCaml syntax tree, load `q_MLast.cmo`.