Camlp5 - Reference Manual

version 4.08

Daniel de Rauglaudre

August 30, 2007

This document was generated by a shell script from the html documentation pages.

Contents

Chapter 1

Introduction

Camlp5 is a preprocessor and pretty-printer for OCaml programs. It also provides parsing and printing tools.

As a preprocessor, it allows to:

- extend the syntax of OCaml,
- and even redefine the whole syntax of the language.

As a pretty printer, it allows to:

- display OCaml programs in an elegant way,
- convert from a syntax to another,
- check the results of syntax extensions.

Camlp5 also provides some parsing and pretty printing tools:

- extensible grammars
- extensible printers
- stream parsers and lexers
- pretty print module

It works as a shell command and can also be used in the OCaml toplevel.

1.1 Shell usage

The main shell commands are:

- `camlp5o` : to treat files written in normal OCaml syntax,
- `camlp5r` : to treat files written in an original syntax named the *revised syntax*.

These commands can be given as parameters of the option `-pp` of the OCaml compiler. Examples:

```
ocamlc -pp camlp5o foo.ml
ocamlc -pp camlp5r bar.ml
```

This way, the parsing is done by Camlp5. In case of syntax errors, the parsing fails with an error message and the compilation is aborted. Otherwise, the OCaml compiler continues with the syntax tree provided by Camlp5.

In the toplevel, it is possible to preprocess the input phrases by loading one of the files "`camlp5o.cma`" or "`camlp5r.cma`". The common usage is:

```
ocaml -I +camlp5 camlp5o.cma
ocaml -I +camlp5 camlp5r.cma
```

1.2 Parsing and Printing kits

Parsing and printing extensions are OCaml object files, i.e. files with the extension "`.cmo`" or "`.cma`". They are the result of the compilation of OCaml source files containing what is necessary to do the parsing or printing. These object files are named parsing and printing *kits*.

These files cannot be linked to produce executables because they generally call functions and use variables defined only in Camlp5 core, typically belonging to the module "`Pcaml`". The kits are destinated to be loaded by the Camlp5 commands, either through their command arguments or through directives in the source files.

It is therefore important to compile the *kits* with the option "`-c`" of the OCaml compiler (i.e. just compilation, not producing an executable) and with the option "`-I +camlp5`" to inform the compiler to find module interfaces in installed Camlp5 library.

In the OCaml toplevel, it is possible to use a kit by simply loading it with the directive "`#load`".

1.3 Extending syntax

A syntax extension is a Camlp5 parsing kit. There are two ways to use a syntax extension:

- Either by giving this object file as parameter to the Camlp5 command. For example:

  ```
  ocamlc -pp "camlp5o ./myext.cmo" foo.ml
  ```

- Or by adding the directive "`#load`" in the source file:

  ```
  #load "./myext.cmo";;
  ```

  and then compile it simply like this:

  ```
  ocamlc -pp camlp5o foo.ml
  ```

Several syntax extensions can be used for a single file. The way to create one's own syntax extensions is explained in the present documentation.

1.4 Pretty printing

Like for syntax extensions, the pretty printing is defined or extended through Camlp5 printing kits. Some pretty printing kits are provided by Camlp5, the main ones being:

- `pr_o.cmo`: to pretty print in normal syntax,
- `pr_r.cmo`: to pretty print in revised syntax.

Examples: if we have a file, `foo.ml`, written in normal syntax and and another one, `bar.ml`, written in revised syntax, here are the commands to pretty print them in their own syntax:

```
camlp5o pr_o.cmo foo.ml
camlp5r pr_r.cmo bar.ml
```

And how to convert them into the other syntax:

```
camlp5o pr_r.cmo foo.ml
camlp5r pr_o.cmo foo.ml
```

The way to create one's own pretty printing extensions is explained in this document.

1.5 Note: the revised syntax

The *revised syntax* is a specific syntax whose aim is to resolve some syntax problems and inconsistencies of the normal OCaml syntax. A chapter will explain the differences between the normal and the revised syntax.

All examples of this documentation are written in that revised syntax. Even if you don't know it, it is not difficult to understand. The same examples can be written in normal syntax. In case of problems, refer to the chapter describing it.

Chapter 2

Parsing and Printing tools

Camlp5 provides two parsing tools:

- stream parsers
- extensible grammars

The first parsing tool, the stream parsers, is the elementary system. It is pure syntactic sugar, i.e. the code is directly converted into basic OCaml statements: essentially functions, pattern matchings, try. A stream parser is just a function. But the system does not manage associativity, nor parsing level. Left recursion results on infinite loops, just like functions whose first action would be a call to itself.

The second parsing tool, the extensible grammars, are more sophisticated. A grammar written with them is more readable, and look like grammars written with tools like "yacc". They take care of associativity, left recursion, and level of parsing. They are dynamically extensible, what allows the syntax extensions what Camlp5 provides for OCaml syntax.

In both cases, the input data are streams.

Camlp5 also provides:

- a pretty printing module
- extensible printers

The next sections give an overview of the parsing and printing tools.

2.1 Stream parsers

The stream parsers is a system of recursive descendant parsing. Streams are actually lazy lists. At each step, the head of the list is compared against a *stream pattern*. There are two kinds of streams parsers:

- The imperative streams parsers, where the elements are removed from the stream as long as they are parsed. Parsers return either:
  - A value, in case of success,
  - The exception "`Stream.Failure`" when the parser does not apply and no elements have been removed from the stream, indicating that, possibly, other parsers may apply,

- The exception "`Stream.Error`" when the parser does not apply, but one or several elements have been removed from the stream, indicating that nothing can to be done to make up the error.

- The purely functional stream parsers where the elements are not removed from the stream during the parsing. These parsers return a value of type "option", i.e either:
  - "Some" a value and the remaining stream, in case of success,
  - "None", in case of failure.

The differences are about:

- Syntax errors: in the imperative version, the location of the error is clear, it is at the current position of the stream, and the system allows to provide a specific error message (typically, that some "element" was "expected"). On the other hand, in the functional version, the position is not clear since it returns nothing and the initial stream is unaffected. The only solution to know where the error happened is to analyze that stream to see how many elements have be unfrozen. No clear error message is available, just "syntax error" (but this could be improved).
- Power: in the imperative version, when a rule raises the exception "`Stream.Error`", the parsing cannot continue. In the functional version, the parsing can continue by analyzing the next rule with the initial unaffected stream: this is a *limited backtrack*.
- Neatness: functional streams are neater, just like functional programming is neater than imperative programming.

The imperative parsers implement what is called "predictive parsing", i.e. recursive descendant parsing without backtrack.

In the imperative version, there exists also lexers, a shorter syntax when the stream elements are of the specific type '`char`'.

2.2 Extensible grammars

Extensible grammars manipulate *grammar entries*. Grammar entries are abstract values internally containing mutable stream parsers. When a grammar entry is created, its internal parser is empty, i.e. it always fails when used. A specific syntactic construction, with the keyword "`EXTEND`" allows to extend grammar entries with new grammar rules.

In opposition to stream parsers, grammar entries manage associativity, left factorization, and levels. Moreover, these grammars allows to define optional calls, lists and lists with separators. However, they are not functions and cannot have parameters.

Since the internal system is stream parsers, extensible grammars use recursive descendant parsing.

The parsers of the OCaml language in Camlp5 are written with extensible grammars.

2.3 Pretty module

The "`Pretty`" module is an original tool allowing to control the displaying of lines. The user has to specify two functions where:

- the data is printed in one only line

- the data is printed in several lines

The system first tries the first function. At any time, it the line overflows, i.e. if its size is greater than some "line length" specified in the module interface, or if it contains newlines, the function is aborted and control is given to the second function.

This is a basic, but powerful, system. It supposes that the programmer takes care of the current indentation, and the beginning and the end of its lines.

The module will be extended in the future to hide the management of indendations and line continuations, and by the supply of functions combinating the two cases above, in which the programmer can specify the possible places where newlines can be inserted.

2.4 Extensible printers

The extensible printers are symmetric to the extensible grammars. The extensible grammars take syntax rules and return syntax trees. The extensible printers are actually extensible functions taking syntax trees as parameters and returning the pretty printed statements in strings.

The extensible printers can have printing levels, just like grammars have parsing levels, and it is possible to take the associativity into account by provided functions to call either the current level or the next level.

The printers of the OCaml language are written with extensible printers.

Part I

Parsing tools

Chapter 3

Stream parsers

We describe here the syntax and the semantics of the parsers of streams of Camlp5. Streams are kinds of lazy lists. The parsers of these streams use recursive descendent method without backtracking, which is the most natural one in functional languages. In particular, parsers are normal functions.

Notice that the parsers have existed in OCaml since many years (the beginning of the 90ies), but some new features have been added in 2007 (lookahead, "no error" optimization, let..in statement and left factorization) in Camlp5 distribution. This chapter describes them also.

3.1 Introduction

Parsers apply to values of type "Stream.t" defined in the module "Stream" of the standard library of OCaml. Like the type "list", the type "Stream.t" has a type parameter, indicating the type of its elements. They differ from the lists that they are lazy (the elements are evaluated as long as the parser need them for its actions), and imperative (parsers deletes their first elements when they take their parsing decisions): notice that purely functional parsers exist in Camlp5, where the corresponding streams are lazy and functional, the analyzed elements remaining in the initial stream and the semantic action returning the resulting stream together with the normal result, which allow natural limited backtrack but have the drawback that it is not easy to find the position of parsing errors when they happen.

Parsers of lazy+imperative streams, which are described here, use a method named "recursive descendent": they look at the first element, they decide what to do in function of its value, and continue the parsing with the remaining elements. Parsers can call other parsers, and can be recursive, like normal functions.

Actually, parsers are just pure syntactic sugar. When writing a parser in the syntax of the parser, Camlp5 transforms them into normal call to functions, use of patterns matchings and try..with statements. The pretty printer of Camlp5, by default, displays this expanded result, without syntax of parsers. A pretty printing kit, when added, can rebuild the parsers in their initial syntax and display it.

3.2 Syntax

The syntax of the parsers, when loading "pa_rp.cmo" (or already included in the command "camlp5r"), is the following:

```
expression ::= parser
             | match-with-parser
    parser ::= "parser" pos-opt "[" parser-cases "]"
```

```
                      | "parser" pos-opt parser-case
   match-with-parser ::= "match" expression "with" parser
        parser-cases ::= parser-cases parser-case
                      | <nothing>
         parser-case ::= "[:" stream-pattern ":]" pos-opt "->" expression
      stream-pattern ::= stream-patt-comp
                      | stream-patt-comp ";" stream-patt-cont
                      | "let" LIDENT "=" expression "in" stream-pattern
                      | <nothing>
    stream-patt-cont ::= stream-patt-comp-err
                      | stream-patt-comp-err ";" stream-patt-cont
                      | "let" LIDENT "=" expression "in" stream-patt-cont
stream-patt-comp-err ::= stream-patt-comp
                      | stream-patt-comp "?" expression
                      | stream-patt-comp "!"
    stream-patt-comp ::= "`" pattern
                      | "`" pattern "when" expression
                      | "?=" lookaheads
                      | pattern "=" expression
                      | pattern
          lookaheads ::= lookaheads "|" lookahead
                      | lookahead
           lookahead ::= "[" patterns "]"
            patterns ::= patterns pattern
                      | pattern
             pos-opt ::= pattern
                      | <nothing>
```

3.3 Streams

The parsers are functions taking streams as parameter. Streams are are values of type "`Stream.t a`" for some type "`a`". It is possible to build streams using the functions defined in the module "`Stream`":

3.3.1 Stream.from

"`Stream.from f`" returns a stream built from the function "`f`". To create a new stream element, the function "`f`" is called with the current stream count, starting with zero. The user function "`f`" must return either "`Some <value>`" for a value or "`None`" to specify the end of the stream.

3.3.2 Stream.of_list

Return a stream built from the list in the same order.

3.3.3 Stream.of_string

Return a stream of the characters of the string parameter.

3.3.4 Stream.of_channel

Return a stream of the characters read from the input channel parameter.

3.4 Semantics of parsers

3.4.1 Parser

A parser, defined with the syntax "parser" above, is of type "`Stream.t a -> b`" where "a" is the type of the elements of the streams and "b" the type of the result. The parser cases are tested in the order they are defined until one of them applies. The result is the semantic action of the parser case which applies. If no parser case applies, the exception "`Stream.Failure`" is raised.

When testing a parser case, if the first stream pattern component matches, all remaining stream pattern components of the stream pattern must match also. If one does not match, the parser raises the exception "`Stream.Error`" which has a parameter of type string: by default, this string is the empty string, but if the stream pattern component which does not match is followed by a question mark and an expression, this expression is evaluated and given as parameter to "`Stream.Error`".

In short, a parser can return with three ways:

- A normal result, of type "`b`" for a parser of type "`Stream.t a -> b`".
- Raising the exception "`Stream.Failure`".
- Raising the exception "`Stream.Error`".

Fundamentally, the exception "`Stream.Failure`" means "this parser does not apply and no element have been removed from the initial stream". This is a normal case when parsing: the parser locally fails, but the parsing can continue.

Conversely, the exception "`Stream.Error`" means that "this parser encountered a syntax error". In this case, the parsing definitively fails.

3.4.2 Left factorization

In parsers, *consecutive* rules starting with the same components are left factorized. It means that they are transformed into one only rule starting with the common path, and continuing with a call to a parser separating the two cases. The order is kept, except that the possible empty rule is inserted at the end.

For example, the parser:

```
parser
[ [: `If; e1 = expr; `Then; e2 = expr; `Else; e3 = expr :] -> f e1 e2 e3
| [: `If; e1 = expr; `Then; e2 = expr :] -> g e1 e2 ]
```

is transformed into:

```
parser
  [: `If; e1 = expr; `Then; e2 = expr;
     a =
       parser
       [ [: `Else; e3 = expr :] -> f e1 e2 e3
       | [: :] -> g e1 e2 ] :] -> a
```

The version where rules are inverted:

```
parser
[ [: `If; e1 = expr; `Then; e2 = expr :] -> g e1 e2
| [: `If; e1 = expr; `Then; e2 = expr; `Else; e3 = expr :] -> f e1 e2 e3 ]
```

is transformed into the same parser.

Notice that:

- Only *consecutive* rules are left factorized. In the following parser:

  ```
  parser
  [ [: `If; e1 = expr; `Then; e2 = expr; `Else; e3 = expr :] -> ...
  | [: a = b :] -> a
  | [: `If; e1 = expr; `Then; e2 = expr :] -> ... ]
  ```

  the two rules starting with "`If`" are not left factorized, and the second "`If`" rule will never work.

- The components which are not *identical* are not factorized. In the following parser:

  ```
  parser
  [ [: `If; e1 = expr; `Then; e2 = expr; `Else; e3 = expr :] -> ...
  | [: `If; e4 = expr; `Then; e2 = expr :] -> ... ]
  ```

  only the first component, "`If`" is factorized, the second one being different because of different patterns ("`e1`" and "`e4`").

3.4.3 Match with parser

The syntax "match expression with parser" allows to match a stream against a parser. It is, for "parser", the equivalent of "match expression with" for "fun". The same way we could say:

```
match expression with ...
```

could be considered as an equivalent to:

```
(fun ...) expression
```

we could consider that:

```
match expression with parser ...
```

is an equivalent to:

```
(parser ...) expression
```

3.4.4 Error messages

A "`Stream.Error`" exception is raised when a stream pattern component does not match and that it is not the first one of the parser case. This exception has a parameter of type string, useful to specify the error message. By default, this is the empty string. To specify an error message, add a question mark and an expression after the stream pattern component. A typical error message is "that stream pattern component expected". Example with the parser of "if..then..else.." above:

```
parser
  [: `If; e1 = expr ? "expression expected after 'if'";
     `Then ? "'then' expected";
     e2 = expr ? "expression expected after 'then'";
     a =
       parser
       [ [: `Else; e3 = expr ? "expression expected" :] -> f e1 e2 e3
       | [: :] -> g e1 e2 ] :] -> a
```

Notice that the expression after the question mark is evaluated only in case of syntax error. Therefore, it can be a complicated call to a complicated function without slowing down the normal parsing.

3.4.5 Stream pattern component

In a stream pattern (starting with "`[:`" and ending with "`:]`"), the stream pattern components are separated with the semicolon character. There are three cases of stream pattern components with some sub-cases for some of them, and an extra syntax can be used with a "let..in" construction. The three cases are:

- A direct test of one or several stream elements (called **terminal** symbol), in three ways:
  1. The character "backquote" followed by a pattern, meaning: if the stream starts with an element which is matched by this pattern, the stream pattern component matches, and the stream element is removed from the stream.
  2. The character "backquote" followed by a pattern, the keyword "when" and an expression of type "`bool`", meaning: if the stream starts with an element which is matched by this pattern and if the evaluation of the expression is "`True`", the stream pattern component matches, and the first element of the stream is removed.
  3. The character "question mark" followed by the character "equal" (new feature 2007) and a lookahead expression (see further), meaning: if the lookahead applies, the stream pattern component matches. The lookahead may unfreeze one or several elements on the stream, but does not remove them.
- A pattern followed by the "equal" sign and an expression of type "`Stream.t x -> y`" for some types "`x`" and "`y`". This expression is called a **non terminal** symbol. It means: call the expression (which is a parser) with the current stream. If this sub-parser:
  1. Returns an element, the pattern is bound to this result and the next stream pattern component is tested.
  2. Raises the exception "`Stream.Failure`", there are two cases:
     - if the stream pattern component is the first one of the stream case, the current parser also fails with the exception "`Stream.Failure`".
     - if the stream pattern component is not the first one of the stream case, the current parser fails with the exception "`Stream.Error`".

     In this second case:
     - If the stream pattern component is followed by a "question mark" and an expression (which must be of type "`string`"), the expression is evaluated and given as parameter of the exception "`Stream.Error`".
     - If the expression is followed by an "exclamation mark" (new feature 2007), the test and conversion from "`Stream.Failure`" to "`Stream.Error`" is not done, and the parser just raises "`Stream.Failure`" again. This is an optimization which must be assumed by the programmer, in general when he knows that the sub-parser called never raises "`Stream.Failure`" (for

example if the called parser ends with a parser case containing an empty stream pattern). See "no error optionization" below.
  - Otherwise the exception parameter is the empty string.

- A pattern, which is bound to the current stream.

Notice that patterns are bound immediately and can be used in the next stream pattern component.

3.4.6 Let statement

Between stream pattern components, it is possible to use the "let..in" construction (new feature 2007). This is not considered as a real stream pattern component, in the fact that is is not tested against the exception "`Stream.Failure`" it may raise. It can be useful for intermediate computation. In particular, it is used internally by the lexers (see chapter about lexers as character stream parsers).

Example of use, when an expression have to be used several times (in the example, "`d a`", which is bound to the variable "`c`"):

```
parser
  [: a = b;
     let c = d a in
     e =
       parser
       [ [: f = g :] -> h c
       | [: :] -> c ] :] -> e
```

3.4.7 Lookahead

The lookahead feature allows to look at several terminals in the stream without removing them, in order to take decisions when more than one terminal is necessary.

For example, when parsing the normal syntax of the OCaml language, there is a problem, in recursing descendent parsing, for the cases where to treat and differentiate the following inputs:

```
(-x+1)
(-)
```

The first case is treated in a rule, telling: "a left parenthesis, followed by an expression, and a right parenthesis". The second one is "a left parenthesis, an operator, a right parenthesis". If programming it like this (left factorizing the first parenthesis):

```
parser
  [: `Lparen;
     e =
       parser
       [ [: e = expr; `Rparen :] -> e
       | [: `Minus; `Rparen :] -> minus_op ] :] -> e
```

this does not work if the input is "`(-)`" because the rule "`e = expr`" accepts the minus sign as expression start, removing it from the input stream and fails as parsing error, while encountering the right parenthesis.

Conversely, writing it this way:

```
parser
  [: `Lparen;
     e =
       parser
       [ [: `Minus; `Rparen :] -> minus_op
       | [: e = expr; `Rparen :] -> e ] :] -> e
```

does not help, because if the input is "`(-x+1)`" the rule above starting with "`` `Minus ``" is accepted and the exception "`Stream.Error`" is raised while encountering the variable "`x`" since a right parenthesis is expected.

In general, this kind of situation is resolved by a left factorization of the parser cases (see the section "Semantics" above), but it is not possible in this case. The solution is to test whether the character after the minus sign is a right parenthesis:

```
parser
  [: `Lparen;
     e =
       parser
       [ [: ?= [ _ Rparen ]; `Minus; `Rparen :] -> minus_op
       | [: e = expr; `Rparen :] -> e ] :] -> e
```

It is possible to put several lists of patterns separated by a vertical bar in the lookahead construction, but with a limitation (due to the implementation): all lists of patterns must have the same number of elements.

3.4.8 No error optimization

The "no error optimization" is a new feature 2007. This is the fact to end a stream pattern component of kind "non-terminal" ("pattern" "equal" "expression") by the character "exclamation mark". Like said above, this inhibits the transformation of the exception "`Stream.Failure`", possibly raised by the called parser, into the exception "`Stream.Error`".

The code:

```
parser [: a = b; c = d ! :] -> e
```

is equivalent to:

```
parser [: a = b; s :] -> let c = d s in e
```

One interest of the first syntax is that it shows to readers that "`d`" is indeed a sub-parser. In the second syntax, it is called in the semantic action, which makes the parser case not no clear, as far as readability is concerned.

If the stream pattern component is the last one of the stream pattern, this allow possible tail recursion done by the OCaml compiler, in the following case:

```
parser [: a = b; c = d ! :] -> c
```

since it is equivalent (with the fact that "`c`" is at the same time the pattern of the last case and the expression of the parser case semantic action:

```
parser [: a = b; s :] -> d s
```

The call to "`d s`" can be a tail recursive call. Without the use of the "exclamation mark" in the rule, the equivalent code is:

```
parser [: a = b; s :] ->
  try d s with [ Stream.Failure -> raise (Stream.Error "") ]
```

which is not tail recursive (because the "try..with" construction pushes a context), preventing the compiler to optimize its code. It can have some importance when many recursive calls happen, since it can overflow the OCaml stack.

3.4.9 Position

The optional "pattern" before and after a stream pattern is bound to the current stream count. Indeed, streams internally contain the count of its elements. At the beginning the count is zero. When an element is removed, the count is incremented. The example:

```
parser [: a = b :] ep -> c
```

is equivalent to:

```
parser [: a = b; s :] -> let ep = Stream.count s in c
```

There is not direct syntax equivalent to the optional pattern at beginning of the stream pattern:

```
parser bp [: a = b :] -> c
```

These optional patterns allow to dispose of the stream count at the beginning and at the end of the parser case, allowing to compute locations of the rule in the source. In particular, if the stream is a stream of characters, these counts are the source location in number of characters.

3.4.10 Semantic action

In a parser case, after the stream pattern, there is an "arrow" and an expression, called the "semantic action". If the parser case is matched the parser returns with the evaluated expression whose environment contains all values bound in the stream pattern.

3.5 Remarks

3.5.1 Simplicity vs Associativity

This parsing technology has the advantage to be simple to use and to understand. But is supposes to sometimes left factorize the rules. Moreover, it does not treat the associativity of operators. For example, if you try to write a parser like this (to compute arithmetic expressions):

```
value rec expr =
  parser
  [ [: e1 = expr; ‘’+’; e2 = expr :] -> e1 + e2
  | [: ‘(’0’..’9’ as c) :] -> Char.code c - Char.code ’0’ ]
```

this would endless loop, exactly like if you wrote a code starting like:

```
value rec expr e =
  let e1 = expr e in
  ...
```

A solution to that is to treat the associavity "by hand", by reading a sub-expression, then looping with a parser case parsing the operator and another sub-expression, and so on.

Another solution is to previously write parsing "combinators". Indeed, parsers being normal functions, it is possible to make a function which takes a parser as parameter and returning a parser using it. For example, left and right associativity parsing combinators:

```
value rec left_assoc op elem =
  let rec op_elem x =
    parser
    [ [: t = op; y = elem; r = op_elem (t x y) :] -> r
    | [: :] -> x ]
  in
  parser [: x = elem; r = op_elem x :] -> r
;

value rec right_assoc op elem =
  let rec op_elem x =
    parser
    [ [: t = op; y = elem; r = op_elem y :] -> t x r
    | [: :] -> x ]
  in
  parser [: x = elem; r = op_elem x :] -> r
;
```

which can be used, e.g. like this:

```
value expr =
  List.fold_right (fun op elem -> op elem)
    [left_assoc (parser [: ''+' :] -> fun x y -> x +. y);
     left_assoc (parser [: ''*' :] -> fun x y -> x *. y);
     right_assoc (parser [: ''^' :] -> fun x y -> x ** y)]
    (parser [: '('0'..'9' as c) :] -> float (Char.code c - Char.code '0'))
;
```

and tested, e.g. in the toplevel, like that:

```
expr (Stream.of_string "2^3^2+1");
```

The same way, it is possible to parser non-context free grammars, by programming parsers returning other parsers.

A third solution, to resolve the problem of associativity, is to use the grammars of Camlp5, which have the other advantage that they are extensible.

3.5.2 Lexing vs Parsing

In general, while analyzing a language, there are two levels:

- The level where the input, considered as a stream of characters, is read to make a stream of tokens (for example "words", if it is a human language, or punctuation). This level is generally called "lexing".
- The level where the input is a stream of tokens where grammar rules are parsed. This level is generally called "parsing".

The ”parser” construction described here can be used for both, thanks to the polymorphism of OCaml:

- The lexing level is a ”parser” of streams of characters returning tokens.
- The parsing level is a ”parser” of streams of tokens returning syntax trees.

By comparison, the programs ”lex” and ”yacc” use two different technologies. With ”parser”s, it is possible to use the same one for both.

3.5.3 Lexer syntax vs Parser syntax

For ”lexers”, i.e. for the specific case of parsers when the input is a stream of characters, it is possible to use a shorter syntax. See the chapter on lexers. They have another syntax, shorter and adapted for the specific type ”`char`”. But they still are internally parsers of streams with the same semantics.

3.5.4 Purely functional parsers

This system of parsers are not purely functional in the sense that the stream structure is imperative: while parsing, the stream advances and the already parsed terminals disappear from the stream structure. This is useful because it is not necessary to return the remaining stream together with the normal result. And it is the reason why there is this `"Stream.Error"` exception: when it happens, it means that some terminals have been consummed from the stream, definitively lost, and that therefore it is no more possible to try other parser cases.

An alternative of that is using purely functional parsers using a new stream type, lazy but not destructive. Their advantage is that they use a limited backtrack: the case of ”if..then..else..” and the shorter ”if..then..” work without having to left factorize the parser cases, and there is no need to lookahead. They have no equivalent to the exception ”`Stream.Error`”: when all cases are tested, and have failed, the parsers return the value ”`None`”. Their drawback is that, when a parsing error happens, it is not easily possible to know the location of the error in the input, since the initial stream has not been modified: the system would indicate a failure at the first character of the first line: this is a general drawback of backtracking parsers. See the solutions found to this problem in the chapter about purely functional parsers.

Chapter 4

Stream lexers

The file "pa_lex.cmo" is a Camlp5 syntax extension kit for parsers of streams of the type 'char'. This syntax is shorter and more readable than its equivalent version written with classical stream parsers. But, like classical parsers, they use recursive descendant parsing. They are also pure syntax sugar, and each lexer written with this syntax can be written using normal parsers syntax.

4.1 Introduction

Classical parsers in OCaml apply to streams of any type of values. For the specific type "char", it has been possible to shorten encoding in several ways, in particular by using strings to group several characters together, and by hidding the management of a "lexing buffer", a data structure recording the matched characters.

Let us take an example. The following function parses an identifier, composed of letters, digits, underscores, quotes and utf-8 bytes, and record the result in a buffer. In classical parsers syntax, this could be written like this:

```
value rec ident buf =
  parser
  [ [: `('A'..'Z' | 'a'..'z' | '0'..'9' | '_' | ''' | '\128'..'\255'
       as c); buf = ident (B.add c buf) ! :] -> buf
  | [: :] -> buf ]
;
```

With the new syntax, it is possible to write it as:

```
value rec ident = lexer [ "A..Za..z0..9_'\128..\255" ident! | ];
```

The two codes are strictly equivalent, but the lexer version is easier to write and understand, and is much shorter.

4.2 Syntax

When loading the syntax extension pa_lex.cmo, the OCaml syntax is extended as follows:

```
expression ::= lexer
     lexer ::= "lexer" "[" rules "]"
     rules ::= rules rule
```

```
                     | <nothing>
                rule ::= symbols [ "->" action ]
             symbols ::= symbols symbol err
                     | <nothing>
              symbol ::= "_" no-record-opt
                     | STRING no-record-opt
                     | simple-expression
                     | "?=" "[" lookaheads "]"
                     | "[" rules "]"
       no-record-opt ::= "/"
                     | <nothing>
   simple-expression ::= LIDENT
                     | CHAR
                     | "(" <expression> ")"
          lookaheads ::= lookaheads "|" lookahead-sequence
                     | lookahead-sequence
  lookahead-sequence ::= lookahead-symbols
                     | STRING
   lookahead-symbols ::= lookahead-symbols lookahead-symbol
                     | lookahead-symbol
    lookahead-symbol ::= CHAR
                     | "_"
                 err ::= "?" simple-expression
                     | "!"
                     | <nothing>
              action ::= expression
```

The identifiers "`STRING`", "`CHAR`" and "`LIDENT`" above represent the OCaml tokens corresponding to string, character and lowercase identifier (identifier starting with a lowercase character).

Moreover, together with that syntax extension, another extension is added the entry `expression`, typically for the semantics actions of the "`lexer`" statement above, but not only. It is:

```
expression ::= "$" "add" STRING
             | "$" "buf"
             | "$" "empty"
             | "$" "pos"
```

Remark: the identifiers "add", "buf", "empty" and "pos" do not become keywords (they are not reserved words) but just identifiers. On the contrary, the identifier "`lexer`", which introduces the syntax, is a new keyword and cannot be used as variable identifier any more.

4.3 Semantics

A lexer defined in the syntax above is a shortcut version of a parser applied to the specific case of streams of characters. It could be written with a normal parser. The proposed syntax is much shorter, easier to use and to understand, and silently takes care of the lexing buffer for the programmer. The lexing buffers are data structures, which are passed as parameters to called lexers and returned by them.

Our lexers are of the type:

```
B.t -> Stream.t char -> u
```

where "u" is a type which depends on what the lexer returns. If there is no semantic action (since it it optional), this type is automatically "B.t".

"B" is a module which must be defined by the user. It has to contain the lexing buffer type "t" and some variables and functions:

- empty: the empty lexing buffer
- add: the way to add a character to a lexing buffer
- get: the way to get a string from the lexing buffer

A possible implementation, using "list char" as lexing buffer type ("B.t"), recording the characters at top of the list (therefore creating a list in reverse order) could be:

```
(* tool function, converting a reversed list of char into a string *)
value rev_implode cl =
  let s = String.create (List.length cl) in
  loop (String.length s - 1) cl where rec loop i =
    fun
    [ [c :: cl] -> do { s.[i] := c; loop (i - 1) cl }
    | [] -> s ]
 ;

 (* the lexing buffer module *)
 module B =
   struct
     type t = list char;
     value empty = [];
     value add c l = [c :: l];
     value get = rev_implode;
   end
;
```

A lexer is a function with two parameters: the first one is the lexing buffer itself, and the second one the stream. When called, it tries to match the stream against its first rule. If it fails, it tries its second rule, and so on, up to its last rule. If the last rule fails, the lexer fails by raising the exception "Stream.Failure". All of this is the usual behaviour of stream parsers.

In a rule, when a character is matched, it is inserted into the lexing buffer, except if the "no record" feature is used (see further).

Rules which have no semantic action return the lexing buffer itself.

4.3.1 Symbols

The different kinds or symbols in a rule are:

- The token "underscore", which represents any character. Fails only if the stream is empty.
- A string which represent a matching of any character in the string. Notice that it is a choice between all the characters, not the sequence of these characters. To indicate a sequence, you have to use several symbols, the ones behind the others. Character ranges can be inserted using the characters "..". For example, to specify a match of any letter or digit, you can write "A..Za..z0..9". Conversely, the

beginning of a comment in the OCaml language has to be written: `"(" "*"`, not `"(*"` which would mean "the character left parenthesis or the character star".

- An expression corresponding to a call to another lexer, which takes the buffer as first parameter and has to return another lexing buffer with all characters found in the stream catened to the lexing buffer.
- The sequence "`?=`" introducing lookahead characters.
- A rule, recursively, between brackets, inlining a lexer.

In the first two cases (namely, underscore and string), the symbol can be optionally followed by the character "slash" specifying that the found symbol must not be added into the lexing buffer. By default, it is. Useful, for example, when writing a lexer parsing strings, when the initial double quote and final double quote have not to be part of the string itself.

Moreover, a symbol can be followed by an optional error indicator, which can be:

- The character `?` (question mark) followed by a string expression, telling that, if there is a syntax error at this point (i.e. the symbol is not matched although the beginning of the rule was), the exception `Stream.Error` is raised with that string as parameter. Without this indicator, it is raised with the empty string. This is the same behaviour than with classical stream parsers.
- The character `!` (exclamation mark), which is just an indicator to let the syntax expander optimize the code. If the programmer is sure that the symbol never fails (i.e. never raises `Stream.Failure`), in particular if this symbol recognizes the empty rule, he can add this exclamation mark. If it is used correctly (the compiler cannot check it), the behaviour is identical as without the `!`, except that the code is shorter and faster, and can sometimes be tail recursive. If the indication is not correct, the behaviour of the lexer is undefined. This feature exists also in classical stream parsers (it is a new feature added in 2007).

4.3.2 Specific expressions

When loading this syntax extension, the entry `<expression>`, at level labelled "simple" of the OCaml language is extended with the following rules:

- `$add` followed by a string, specifing that the programmer wants to add all characters of the string in the lexing buffer. It returns the new lexing buffer. It corresponds to an iteration of calls to `B.add` with all characters of the string with the current lexing buffer as initial parameter.
- `$buf` which returns the lexing buffer converted into string.
- `$empty` which returns an empty lexing buffer.
- `$pos` which return the current position in the stream in number of characters (starting at zero).

4.3.3 Lookahead

Lookahead is useful in some cases, when factorization of rules is impossible. To understand how it is useful, a first remark must be done, about the usual behaviour of Camlp5 stream parsers.

Stream parsers (including these lexers) use a limited parsing algorithm, in a way that when the first symbol of a rule is matched, all the following symbols of the same rule must apply, otherwise it is a syntax error. There is no backtrack. In most of the cases, left factorization of rules resolve conflicting problems. For example, in parsers of tokens (which is not our case here, since we parse only characters), when one writes a parser to recognize both the typicall grammar rules "if..then..else" and the shorter "if..then..", the solution is to write a rule starting with "if..then.." followed with a call to a parser recognizing "else.." or nothing.

Sometimes, however, this left factorization is not possible. A lookahead of the stream to check the presence of some elements (these element being characters, if we are using this "lexer" syntax) might be necessary to decide whether or not it is a good idea to start the rule. This lookahead feature may unfreeze several characters from the input stream but without removing them.

Syntactically, a lookahead starts with `?=` and is followed by one or several lookahead sequences separated by the vertical bar `|`, the whole list being enclosed by braces.

If there are several lookaheads, they must all be of the same size (contain the same number of characters).

If the lookahead sequence is just a string, it corresponds to all characters of this string in the order (which is different for strings outside lookahead sequences, representing a choice of all characters).

Examples of lookaheads:

```
?= [ _ ''' | '\\' _ ]
?= [ "<<" | "<:" ]
```

The first line above matches a stream whose second character is a quote or a stream whose first character is a backslash (real example in the lexer of OCaml, in the library of Camlp5, named "plexer.ml"). The second line matches a stream starting with the two characters `<` and `<` or starting with the two characters `<` and `:` (this is another example in the same file).

4.3.4 Semantic actions of rules

By default, the result of a "lexer" is the current lexing buffer, which is of type "`B.t`". But it is possible to return other values, by adding "`->`" at end of rules followed by the expression you want to return, like in usual pattern matching in OCaml.

An interesting result, for example, could be the string corresponding to the characters of the lexing buffer. This can be obtained by returning the value "`$buf`".

4.3.5 A complete example

A complete example can be seen in the sources of Camlp5, file "lib/plexer.ml". This is the lexer of OCaml, either "normal" or "revised" syntax.

4.3.6 Compiling

To compile a file containing lexers, just load `pa_lex.cmo` using one of the following methods:

- Either by adding `pa_lex.cmo` among the Camlp5 options. See the Camlp5 manual page or documentation.
- Or by adding `#load "pa_lex.cmo";` anywhere in the file, before the usages of this "lexer" syntax.

4.3.7 How to display the generated code

You can see the generated code, for a file "bar.ml" containing lexers, by typing in a command line:

```
camlp5r pa_lex.cmo pr_r.cmo bar.ml
```

To see the equivalent code with stream parsers, use:

```
camlp5r pa_lex.cmo pr_r.cmo pr_rp.cmo bar.ml
```

Chapter 5

Functional parsers

Purely functional parsers are an alternative of stream parsers where the used stream type is a lazy non-destructive type : these streams are lazy values, like in classical stream parsers, but the values are not removed as long as the parsing advances. To make them work, the parsers of purely functional streams return, not the simple values, but a value of type option : "`Node`" meaning "mo match" (the equivalent of the exception "`Parse.Failure`" of normal streams) and "`Some (r, s)`" meaning "the result is r and the remaining stream is s".

5.1 Syntax

The syntax of purely functional parsers, when loading "pa_fstream.cmo", is the following:

```
        expression ::= fparser
                     | match-with-fparser
           fparser ::= "fparser" pos-opt "[" parser-cases "]"
                     | "fparser" pos-opt parser-case
 match-with-fparser ::= "match" expression "with" fparser
      parser-cases ::= parser-cases parser-case
                     | <nothing>
       parser-case ::= "[:" stream-pattern ":]" pos-opt "->" expression
                     | "[:" ":]" pos-opt "->" expression
    stream-pattern ::= stream-patt-comp
                     | stream-patt-comp ";" stream-pattern
  stream-patt-comp ::= "`" pattern
                     | pattern "=" expression
                     | pattern
           pos-opt ::= pattern
                     | <nothing>
```

Notice that, in difference with classical parsers, there is no difference, in a stream pattern, between the first stream pattern component and the other ones. In particular, there is not this syntax "question mark" and expression optionnally ending those components. Moreover, the "lookahead" case is not necessary, we see further why. On the contrary, the syntaxes "pattern when" and "let..in" inside stream patterns we see in classical parsers are just not implemented.

5.2 Streams

The functional parsers are functions taking as parameters functional streams, which are values of type ”`Fstream.t a`” for some type ”`a`”. It is possible to build functional streams using the functions defined in the module ”`Fstream`”:

5.2.1 Fstream.from

”`Fstream.from f`” returns a stream built from the function ”`f`”. To create a new stream element, the function ”`f`” is called with the current stream count, starting with zero. The user function ”`f`” must return either ”`Some <value>`” for a value or ”`None`” to specify the end of the stream.

5.2.2 Fstream.of_list

Return a stream built from the list in the same order.

5.2.3 Fstream.of_string

Return a stream of the characters of the string parameter.

5.2.4 Fstream.of_channel

Return a stream of the characters read from the input channel parameter.

5.3 Semantics of parsers

5.3.1 Fparser

The purely functional parsers act like classical parsers, with a recursive descent algorithm, except that:

- If the first stream pattern component matches the beginning of the stream, there is no error if the following stream patterns components do not match: the control simply passes to the next parser case with the initial stream.
- If the semantic actions are of type ”`t`”, the result of the parser is of type ”`option (t * Fstream.t)`”, not just ”`t`” like in classical parsers. If a stream pattern matches, the semantic action is evaluated, giving some result ”`e`” and the result of the parser is ”`Some (e, strm)`” where ”`strm`” is the remaining stream.
- If no parser case matches, the result of the parser is ”`None`”.

5.3.2 Error position

A difficulty, with purely functional parsers, is how to find the position of the syntax error, when the input is wrong. Since the system tries all parsers cases before returning ”`None`”, and that the initial stream is not affected, it is not possible to directly find where the error happened. This is a problem for parsing using backtracking (here, it is limited backtracking, but the problem is the same).

The solution is to use the function ”`Fstream.count_unfrozen`” applied to the initial stream. Like its name says, it returns the number of unfrozen elements of the stream, which is exactly the longuest match found. If the input is a stream of characters, the return of this function is exactly the position in number of characters from the beginning of the stream.

However, it is not possible to know directly which rule failed and therefore it is not possible, like in classical parsers, to specify and get clear error messages. Future versions of purely functional parsers may propose solutions to resolve this problem.

Notice that, if using that method, it is not possible to reuse the same stream to call another parser, and hope to get the right position of the error, if another error happens, since it may test less terminals than the first parser. Use a fresh stream in this case, if possible.

Chapter 6

Extensible grammars

This chapter describes the syntax and semantics of the extensible grammars of Camlp5.

The extensible grammars are the most advanced parsing tool of Camlp5. They apply to streams of characters using a lexer which has to be previously defined by the programmer. In Camlp5, the syntax of the OCaml language is defined with extensible grammars, which makes Camlp5 a bootstrapped system (it compiles its own features by itself).

6.1 Getting started

The extensible grammars are a system to build *grammar entries* which can be extended dynamically. A grammar entry is an abstract value internally containing a stream parser. The type of a grammar entry is `"Grammar.Entry.e t"` where `"t"` is the type of the values returned by the grammar entry.

To start with extensible grammars, it is necessary to build a *grammar*, a value of type "`Grammar.g`", using the function "`Grammar.gcreate`":

```
value g = Grammar.gcreate lexer;
```

where "`lexer`" is a lexer previously defined. See the section explaining the interface with lexers. In a first time, it is possible to use a lexer of the module "`Plexer`" provided by Camlp5:

```
value g = Grammar.gcreate (Plexer.gmake ());
```

Each grammar entry is associated with a grammar. Only grammar entries of the same grammar can call each other. To create a grammar entry, one has to use the function "`Grammar.Entry.create`" with takes the grammar as first parameter and a name as second parameter. This name is used in case of syntax errors. For example:

```
value exp = Grammar.Entry.create g "expression";
```

To apply a grammar entry, the function "`Grammar.Entry.parse`" can be used. Its first parameter is the grammar entry, the second one a stream of characters:

```
Grammar.Entry.parse exp (Stream.of_string "hello");
```

But if you experiment this, since the entry was just created without any rules, you receive an error message:

```
Stream.Error "entry [expression] is empty"
```

To add grammar rules to the grammar entry, it is necessary to *extend* it, using a specific syntactic statement: "`EXTEND`".

6.2 Syntax of the EXTEND statement

The "`EXTEND`" statement is added in the expressions of the OCaml language when the syntax extension kit "`pa_extend.cmo`" is loaded. Its syntax is:

```
  expression ::= extend
      extend ::= "EXTEND" extend-body "END"
 extend-body ::= global-opt entries
  global-opt ::= "GLOBAL" ":" entry-names ";"
               | <nothing>
 entry-names ::= entry-name entry-names
               | entry-name
       entry ::= entry-name ":" position-opt "[" levels "]"
position-opt ::= "FIRST"
               | "LAST"
               | "BEFORE" label
               | "AFTER" label
               | "LEVEL" label
               | <nothing>
      levels ::= level "|" levels
               | level
       level ::= label-opt assoc-opt "[" rules "]"
   label-opt ::= label
               | <nothing>
   assoc-opt ::= "LEFTA"
               | "RIGHTA"
               | "NONA"
               | <nothing>
       rules ::= rule "|" rules
               | rule
        rule ::= psymbols-opt "->" expression
               | psymbols-opt
psymbols-opt ::= psymbols
               | <nothing>
    psymbols ::= psymbol ";" psymbols
               | psymbol
     psymbol ::= symbol
               | pattern "=" symbol
      symbol ::= keyword
               | token
               | token string
               | entry-name
               | entry-name "LEVEL" label
               | "SELF"
               | "NEXT"
               | "LIST0" symbol
               | "LIST0" symbol "SEP" symbol
               | "LIST1" symbol
               | "LIST1" symbol "SEP" symbol
               | "OPT" symbol
               | "FLAG" symbol
               | "[" rules "]"
```

```
              | "(" symbol ")"
     keyword ::= string
       token ::= uident
       label ::= string
  entry-name ::= qualid
      qualid ::= qualid "." qualid
               | uident
               | lident
      uident ::= 'A'-'Z' ident
      lident ::= ('a'-'z' | '_' | utf8-byte) ident
       ident ::= ident-char*
  ident-char ::= ('a'-'a' | 'A'-'Z' | '0'-'9' | '_' | ''' | utf8-byte)
   utf8-byte ::= '\128'-'\255'
```

Other statements, "GEXTEND", "DELETE_RULE", "GDELETE_RULE" are also defined by the same syntax extension kit. See further.

In the description above, ony "EXTEND" and "END" are new keywords (reserved words which cannot be used in variables, constructors or module names). The other strings (e.g. "GLOBAL", "LEVEL", "LIST0", "LEFTA", etc.) are not reserved.

6.3 Semantics of the EXTEND statement

The EXTEND statement starts with the "EXTEND" keyword and ends with the "END" keyword.

6.3.1 GLOBAL indicator

After the first keyword, it is possible to see the identifier "GLOBAL" followed by a colon, a list of entries names and a semicolon. It says that these entries correspond to visible (previously defined) entry variables, in the context of the EXTEND statement, the other ones being locally and silently defined inside.

- If an entry, which is extended in the EXTEND statement, is in the GLOBAL list, but is not defined in the context of the EXTEND statement, the OCaml compiler will fail with the error "unbound value".
- If there is no GLOBAL indicator, and an entry, which is extended in the EXTEND statement, is not defined in the contex of the EXTEND statement, the OCaml compiler will also fail with the error "unbound value".

Example:

```
value exp = Grammar.Entry.create g "exp";
EXTEND
  GLOBAL: exp;
  exp: [ [ x = foo; y = bar ] ];
  foo: [ [ "foo" ] ];
  bar: [ [ "bar" ] ];
END;
```

The entry "exp" is an existing variable (defined by value exp = ...). On the other hand, the entries "foo" and "bar" have not been defined. Because of the GLOBAL indicator, the system define them locally.

Without the GLOBAL indicator, the three entries would have been considered as global variables, therefore the OCaml compiler would say "unbound variable" under the first undefined entry, "foo".

6.3.2 Entries list

Then the list of entries extensions follow. An entry extension starts with the entry name followed by a colon. An entry may have several levels corresponding to several stream parsers which call the ones the others (see further).

Optional position

After the colon, it is possible to specify a where to insert the defined levels:

- The identifier "`FIRST`" (resp. "`LAST`") indicates that the level must be inserted before (resp. after) all possibly existing levels of the entry. They become their first (resp. last) levels.
- The identifier "`BEFORE`" (resp. "`AFTER`") followed by a level label (a string) indicates that the levels must be inserted before (resp. after) that level, if it exists. If it does not exist, the extend statement fails at run time.
- The identifier "`LEVEL`" followed by a level label indicates that the first level defined in the extend statement must be inserted at the given level, extending and modifying it. The other levels defined in the statement are inserted after this level, and before the possible levels following this level. If there is no level with this label, the extend statement fails at run time.
- By default, if the entry has no level, the levels defined in the statement are inserted in the entry. Otherwise the first defined level is inserted at the first level of the entry, extending or modifying it. The other levels are inserted afterwards (before the possible second level which may previously exist in the entry).

Levels

After the optional "position", the *level* list follow. The levels are separated by vertical bars, the whole list being between brackets.

A level starts with an optional label, which corresponds to its name. This label is useful to specify this level in case of future extensions, using the *position* (see previous section) or for possible direct calls to this specific level.

The level continues with an optional associativity indicator, which can be:

- LEFTA for left associativity (default),
- RIGHTA for right associativity,
- NONA for no associativity.

Rules

At last, the grammar *rule* list appear. The rules are separated by vertical bars, the whole list being brackets.

A rule looks like a match case in the "`match`" statement or a parser case in the "`parser`" statement: a list of psymbols (see next paragraph) separated by semicolons, followed by a right arrow and an expression, the semantic action. Actually, the right arrow and expression are optional: in this case, it is equivalent to an expression which would be the unit "`()`" constructor.

A psymbol is either a pattern, followed with the equal sign and a symbol, or by a symbol alone. It corresponds to a test of this symbol, whose value is bound to the pattern if any.

Symbols

A symbol is either:

- a keyword (a string): the input must match this keyword,
- a token name (an identifier starting with an uppercase character), optionally followed by a string: the input must match this token (any value if no string, or that string if a string follows the token name), the list of the available tokens depending on the associated lexer (the list of tokens available with "Plexer.gmake ()" is: LIDENT, UIDENT, TILDEIDENT, TILDEIDENTCOLON, QUESTIONIDENT, INT, INT_l, INT_L, INT_n, FLOAT, CHAR, STRING, QUOTATION, ANTIQUOT and EOI; other lexers may propose other lists of tokens),
- an entry name, which correspond to a call to this entry,
- an entry name followed by the identifier "`LEVEL`" and a level label, which correspond to the call to this entry at that level,
- the identifier "`SELF`" which is a recursive call to the present entry, according to the associativity (i.e. it may be a call at the current level, to the next level, or to the top level of the entry): "`SELF`" is equivalent to the name of the entry itself,
- the identifier "`NEXT`", which is a call to the next level of the current entry,
- a left brace, followed by a list of rules separated by vertical bars, and a right brace: equivalent to a call to an entry, with these rules, inlined,
- a meta symbol (see further),
- a symbol between parentheses.

The syntactic analysis follow the list of symbols. If it fails, depending on the first items of the rule (see the section about the kind of grammars recognized):

- the parsing may fail by raising the exception "`Stream.Error`"
- the parsing may continue with the next rule.

Meta symbols

Extra symbols exist, allowing to manipulate lists or optional symbols. They are:

- LIST0 followed by a symbol: this is a list of this symbol, possibly empty,
- LIST0 followed by a symbol, SEP and another symbol: this is a list, possibly empty, of the first symbol separated by the second one,
- LIST1 followed by a symbol: this is a list of this symbol, with at least one element,
- LIST0 followed by a symbol, SEP and another symbol: this is a list, with at least one element, of the first symbol separated by the second one,
- OPT followed by a symbol: equivalent to "this symbol or nothing" returning a value of type "`option`".
- FLAG followed by a symbol: equivalent to "this symbol or nothing", returning a boolean.

6.3.3 Rules insertion

Remember that "EXTEND" is a statement, not a declaration: the rules are added in the entries at run time. Each rule is internally inserted in a tree, allowing the left factorization of the rule. For example, with this list of rules (borrowed from the Camlp5 sources):

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

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

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

This tree is built as long as rules are inserted. When used, by applying the function "Grammar.Entry.parse" to the current entry, the input is matched with that tree, starting from the tree root, descending on it as long as the parsing advances.

There is a different tree by entry level.

6.3.4 Semantic action

The semantic action, i.e. the expression following the right arrow in rules, contain in its environment:

- the variables bound by the patterns of the symbols found in the rules,

- the specific variable "loc" which contain the location of the whole rule in the source.

The location is an abstract type defined in the module "Stdpp" of Camlp5.

It is possible to change the name of this variable by using the option "-loc" of Camlp5. For example, compiling a file like this:

```
camlp5r -loc foobar file.ml
```

the variable name, for the location will be "foobar" instead of "loc".

6.4 The DELETE_RULE statement

The "DELETE_RULE" statement is also added in the expressions of the OCaml language when the syntax extension kit "pa_extend.cmo" is loaded. Its syntax is:

```
        expression ::= delete-rule
       delete-rule ::= "DELETE_RULE" delete-rule-body "END"
  delete-rule-body ::= entry-name ":" symbols
           symbols ::= symbol symbols
                     | symbol
```

See the syntax of the EXTEND statement for the meaning of the syntax entries not defined above.

The entry is scanned for a rule matching the giving symbol list. When found, the rule is removed. If no rule is found, the exception "Not_found" is raised.

6.5 Extensions FOLD0 and FOLD1

When loading "pa_extfold.cmo" after "pa_extend.cmo", the entry "symbol" of the EXTEND statement is extended with what is named the *fold iterators*, like this:

```
       symbol ::= "FOLD0" simple_expr simple_expr symbol
                | "FOLD1" simple_expr simple_expr symbol
                | "FOLD0" simple_expr simple_expr symbol "SEP" symbol
                | "FOLD1" simple_expr simple_expr symbol "SEP" symbol
  simple_expr ::= expr (level "simple")
```

Like their equivalent with the lists iterators: "LIST0", "LIST1", "LIST0SEP", "LIST1SEP", they read a sequence of symbols, possibly with the separators, but instead of building the list of these symbols, apply a fold function to each symbol, starting at the second "expr" (which must be a expression node) and continuing with the first "expr" (which must be a function taking two expressions and returing a new expression).

The list iterators can be seen almost as a specific case of these fold iterators where the initial "expr" would be:

```
<:expr< [] >>
```

and the fold function would be:

```
fun e1 e2 -> <:expr< [$e1$ :: $e2$ ] >>
```

except that, implemented like that, they would return the list in reverse order.

Actually, a program using them can be written with the lists iterators with the semantic action applying the function "`List.fold_left`" to the returned list, except that with the fold iterators, this operation is done as long as the symbols are read on the input, no intermediate list being built.

Example, file "sum.ml":

```
#load "pa_extend.cmo";
#load "pa_extfold.cmo";
#load "q_MLast.cmo";
let loc = Stdpp.dummy_loc in
EXTEND
  Pcaml.expr:
    [ [ "sum";
        e =
          FOLD0 (fun e1 e2 -> <:expr< $e2$ + $e1$ >>) <:expr< 0 >>
            Pcaml.expr SEP ";";
        "end" -> e ] ]
  ;
END;
```

which can be compiled like this:

```
ocamlc -pp camlp5r -I +camlp5 -c sum.ml
```

and tested:

```
  ocaml -I +camlp5 camlp5r.cma sum.cmo
          Objective Caml version ...

          Camlp5 Parsing version ...

  # sum 3;4;5 end;
- : int = 12
```

6.6 Extensions SLIST0, SLIST1, SOPT and SFLAG

The parsing kit "`pa_extend_m.cmo`" adds the specific iterators "`SLIST0`", "`SLIST1`", "`SOPT`" and "`SFLAG`". They are used in the file "q_MLast.ml". They allow to generate rules for antiquotations of kind "list" and "opt".

They are not supposed to be used by the programmer.

For information:

The symbol "`SLIST0 symb`" is equivalent to the rule symbol:

```
[ a = a_list -> a
| a = LIST0 symb -> Qast.List a ]
```

Same for the other specific iterators.

The entry "`a_list`" and the constructor "`Qast.List`" are locally defined in "q_MLast.ml". This system allows the updating of the source file "q_MLast.ml" (syntax tree quotations, with antiquotations, in revised syntax) from the file "pa_r.ml" (revised syntax). These grammars are close the one to the other, except that "q_MLast.ml" can parse antiquotations, what is done by replacing the list iterators by these specific iterators. This operation is done through the shell script "mk_q_MLast.sh" in the "meta" directory of the Camlp5 sources.

6.7 Grammar machinery

We explain here the detail of the mechanism of the parsing of an entry.

6.7.1 Start and Continue

At each entry level, the rules are separated into two trees:

- The tree of the rules *not* starting with the current entry name nor by "`SELF`".
- The tree of the rules starting with the current entry name or by the identifier "`SELF`", this symbol not being included in the tree.

They determine two functions:

- The function named "start", analyzing the first tree.
- The function named "continue", taking, as parameter, a value previously parsed, and analyzing the second tree.

A call to an entry, using "`Grammar.Entry.parse`" correspond to a call to the "start" function of the first level of the entry.

The "start" function tries its associated tree. If it works, it calls the "continue" function of the same level, giving the result of "start" as parameter. If this "continue" function fails, this parameter is simply returned. If the "start" function fails, the "start" function of the next level is tested. If there is no more levels, the parsing fails.

The "continue" function first tries the "continue" function of the next level. If it fails, or if it is the last level, it tries its associated tree, then calls itself again, giving the result as parameter. If its associated tree fails, it returns its extra parameter.

6.7.2 Associativity

While testing the tree, there is a special case for rules ending with SELF or with the current entry name. For this last symbol, there is a call to the "start" function: of the current level if the level is right associative, or of the next level otherwise.

There is no behaviour difference between left and non associative, because, in case of syntax error, the system attempts, anyway, to recover the error by applying the "continue" function of the previous symbol (if this symbol is a call to an entry).

When a SELF or the current entry name is encountered in the middle of the rule (i.e. if it is not the last symbol), there is a call to the "start" function of the first level of the current entry.

Example. Let us consider the following grammar:

```
EXTEND
  expr:
    [ "minus" LEFTA
      [ x = SELF; "-"; y = SELF -> x -. y ]
    | "power" RIGHTA
      [ x = SELF; "**"; y = SELF -> x ** y ]
    | "simple"
      [ "("; x = SELF; ")" -> x
      | x = INT -> float_of_int x ] ]
  ;
END
```

The left "SELF"s of the two levels "minus" and "power" correspond to a call to the next level. In the level "minus", the right "SELF" also, and the left associativity is treated by the fact that the "continue" function is called (starting with the keyword "-" since the left "SELF" is not part of the tree). On the other hand, for the level "power", the right "SELF" corresponds to a call to the current level, i.e. the level "power" again. At end, the "SELF" between parentheses of the level "simple" correspond to a call to the first level, namely "minus" in this grammar.

6.7.3 Errors and recovery

Like for stream parsers, two exceptions may happen: "Stream.Failure" or "Stream.Error". The first one indicates that the parsing just could not start. The second one indicates that the parsing started but failed further.

In stream parsers, when the first symbol of a rule has been accepted, all the symbols of the same rule must be accepted, otherwise the exception "Stream.Error" is raised.

Here, in extensible grammars, unlike stream parsers, before the "Stream.Error" exception, the system attempts to recover the error by the following trick: if the previous symbol of the rule was a call to another entry, the system calls the "continue" function of that entry, which may resolve the problem.

In extensible grammars, the exceptions are encapsulated with the exception "Stdpp.Exc_located" giving the location of the error together with the exception itself.

6.7.4 Tokens starting rules

Another improvement (than the error recovery) is the fact that, when a rule starts with several tokens and/or keywords, all these tokens and keywords are tested in one time, and the possible "Stream.Error" may happen, only from the symbol following them on, if any.

6.7.5 Kind of grammar

The kind of grammar is predictive parsing grammar, i.e. recursive descent parsing without backtrack. But with some nuances, due to the improvements (error recovery and token starting rules) indicated in the previous sections.

6.8 The Grammar module

See its section in the chapter "Library".

6.9 Interface with the lexer

To create a grammar, the function "`Grammar.gcreate`" must be called, with a lexer as parameter.

A simple solution, as possible lexer, is the predefined lexer built by "`Plexer.gmake ()`", lexer used for the OCaml grammar of Camlp5. In this case, you can just put it as parameter of "`Grammar.gcreate`" and it is not necessary to read this section.

The section first introduces the notion of "token patterns" which are the way the tokens and keywords symbols in the EXTEND statement are represented. Then follow the description of the type of the parameter of "`Grammar.gcreate`".

6.9.1 Token patterns

A token pattern is a value of the type defined like this:

```
type pattern = (string * string);
```

This type represents values of the token and keywords symbols in the grammar rules.

For a token symbol in the grammar rules, the first string is the token constructor name (starting with an uppercase character), the second string indicates whether the match is "any" (the empty string) or some specific value of the token (an non-empty string).

For a keyword symbol, the first string is empty and the second string is the keyword itself.

For example, given this grammar rule:

```
"for"; i = LIDENT; "="; e1 = SELF; "to"; e2 = SELF
```

the different symbols and keywords are represented by the following couples of strings:

- the keyword "for" is represented by `("", "for")`,
- the keyword "=" by `("", "=")`,
- the keyword "to" by `("", "to")`),
- and the token symbol `LIDENT` by `("LIDENT", "")`.

The symbol `UIDENT "Foo"` in a rule would be represented by the token pattern:

```
("UIDENT", "Foo")
```

Notice that the symbol "`SELF`" is a specific symbol of the EXTEND syntax: it does not correspond to a token pattern and is represented differently. A token constructor name must not belong to the specific symbols: SELF, NEXT, LIST0, LIST1, OPT and FLAG.

6.9.2 The glexer record

The type of the parameter of the function "`Grammar.gcreate`" is "`glexer`", defined in the module "`Token`". It is a record type with the following fields:

`tok_func`

It is the lexer itself. Its type is:

```
Stream.t char -> (Stream.t (string * string) * location_function);
```

The lexer takes a character stream as parameter and must answer a couple of: a token stream, the tokens being represented by a couple of strings, and a location function.

The location function is a function taking, as parameter, a integer corresponding to a token number in the stream (starting from zero), and returning the location of this token in the source. It is important to get the good locations in the semantic actions of the grammar rules.

Notice that, despite the lexer takes a character stream as parameter, it is not mandatory to use the stream parsers technology to write the lexer. What is important is that it does the job.

`tok_using`

It is a function of type:

```
pattern -> unit
```

The parameter of this function is the representation of a token symbol or a keyword symbol in grammar rules. See the section about token patterns.

This function is called for each token symbol and each keyword encountered in the grammar rules of the EXTEND statement. Its goal is to allow the lexer to check that the tokens and keywords do respect the lexer rules. It checks that the tokens exist and are not mispelled. It can be also used to enter the keywords in the lexer keyword tables.

Setting it as the function that does nothing is possible, but the check of correctness of tokens is not done.

In case or error, the function must raise the exception "`Token.Error`" with an error message as parameter.

`tok_removing`

It is a function of type:

```
pattern -> unit
```

It is possibly called by the DELETE_RULE statement for tokens and keywords no more used in the grammar. The grammar system maintains a number of usages of all tokens and keywords and call this function only when this number reaches zero. This can be interesting for keywords: the lexer can remove them from its tables.

`tok_match`

It is a function of type:

```
pattern -> ((string * string) -> unit)
```

The function tells how a token of the input stream is matched against a token pattern. Both are represented by a couple of strings.

This function takes a token pattern as parameter and return a function matching a token, returning the matched string or raising the exception "`Stream.Failure`" if the token does not match.

Notice that, for efficiency, it is necessary to write this function as a match of token patterns returning, for each case, the function which matches the token, *not* a function matching the token pattern and the token together and returning a string for each case.

An acceptable function is provided in the module "Token" and is named "default_match". Its code looks like this:

```
value default_match =
  fun
  [ (p_con, "") ->
      fun (con, prm) -> if con = p_con then prm else raise Stream.Failure
  | (p_con, p_prm) ->
      fun (con, prm) ->
        if con = p_con \&\& prm = p_prm then prm else raise Stream.Failure ]
;
```

`tok_text`

It is a function of type:

```
pattern -> string
```

Destinated to error messages, it takes a token pattern as parameter and return the string giving its name.

It is possible to use the predefined function "`lexer_text`" of the Token module. This function just returns the name of the token pattern constructor and its parameter if any.

For example, with this default function, the token symbol IDENT would be written as IDENT in error message (e.g. "IDENT expected"). The "text" function may decide to print it differently, e.g., as "identifier".

`tok_comm`

It is a mutable field of type:

```
option (list location)
```

It asks the lexer (the lexer function should do it) to record the locations of the comments in the program. Setting this field to "None" indicates that the lexer must not record them. Setting it to "Some []" indicated that the lexer must put the comments location list in the field, which is mutable.

6.9.3 Minimalist version

If a lexer have been written, named "`lexer`", here is the minimalist version of the value suitable as parameter to "`Grammar.gcreate`":

```
{Token.tok_func = lexer;
 Token.tok_using _ = (); Token.tok_removing _ = ();
 Token.tok_match = Token.default_match;
 Token.tok_text = Token.lexer_text;
 Token.tok_comm = None}
```

6.10 Functorial interface

The normal interface for grammars described in the previous sections has two drawbacks:

- First, the type of tokens of the lexers must be "`(string * string)`"
- Second, since the entry type has no parameter to specify the grammar it is bound to, there is no static check that entries are compatible, i.e. belong to the same grammar. The check is done at run time.

The functorial interface resolve these two problems. The functor takes a module as parameter where the token type has to be defined, together with the lexer returning streams of tokens of this type. The resulting module define entries compatible the ones to the other, and this is controlled by the OCaml type checker.

The syntax extension must be done with the statement GEXTEND, instead of EXTEND, and deletion by GDELETE_RULE instead of DELETE_RULE.

6.10.1 The glexer type

In the section about the interface with the lexer, we presented the glexer type as a record without type parameter. Actually, this type is defined as:

```
type glexer 'te =
  { tok_func : lexer_func 'te;
    tok_using : pattern -> unit;
    tok_removing : pattern -> unit;
    tok_match : pattern -> 'te -> string;
    tok_text : pattern -> string;
    tok_comm : mutable option (list location) }
;
```

where the type parameter is the type of the token, which can be any type, different from "`(string * string)`", providing the lexer function (`tok_func`) returns a stream of this token type and the match function (`tok_match`) indicates how to match values of this token type against the token patterns (which remain defined as "`(string * string)`").

Here is an example of an user token type and the associated match function:

```
type mytoken =
  [ Ident of string
  | Int of int
  | Comma | Equal
  | Keyw of string  ]
;

value mymatch =
  fun
  [ ("IDENT", "") ->
      fun [ Ident s -> s | _ -> raise Stream.Failure ]
  | ("INT", "") ->
      fun [ Int i -> string_of_int i | _ -> raise Stream.Failure ]
  | ("", ",") ->
      fun [ Comma -> "" | _ -> raise Stream.Failure ]
  | ("", "=") ->
```

```
      fun [ Equal -> "" | _ -> raise Stream.Failure ]
  | ("", s) ->
      fun
      [ Keyw k -> if k = s then "" else raise Stream.Failure
      | _ -> raise Stream.Failure ]
  | _ -> raise (Token.Error "bad token in match function") ]
;
```

6.10.2 The functor parameter

The type of the functor parameter is defined as:

```
module type GLexerType =
  sig
    type te = 'x;
    value lexer : Token.glexer te;
  end;
```

The token type must be specified (type "te") and the lexer also, with the interface for lexers, of the glexer type defined above, the record fields being described in the section "interface with the lexer", but with a general token type.

6.10.3 The resulting grammar module

Once a module of type "GLexerType" has been built (previous section, it is possible to create a grammar module by applying the functor "Grammar.GMake". For example:

```
module MyGram = Grammar.GMake MyLexer;
```

Notice that the function "Entry.parse" of this resulting module does not take a character stream as parameter, but a value of type "parsable". This function is equivalent to the function "parse_parsable" of the non functorial interface. In short, the parsing of some character stream "cs" by some entry "e" of the example grammar above, must be done by:

```
MyGram.Entry.parse e (MyGram.parsable cs)
```

instead of:

```
MyGram.Entry.parse e cs
```

6.10.4 GEXTEND and GDELETE_RULE

The "GEXTEND" and "GDELETE_RULE" statements are also added in the expressions of the OCaml language when the syntax extension kit "pa_extend.cmo" is loaded. They have to be used for grammars defined with the functorial interface. Their syntax are:

```
          expression ::= gextend
                       | gdelete-rule
        gdelete-rule ::= "GDELETE_RULE" gdelete-rule-body "END"
             gextend ::= "GEXTEND" gextend-body "END"
        gextend-body ::= grammar-module-name extend-body
   gdelete-rule-body ::= grammar-module-name delete-rule-body
 grammar-module-name ::= qualid
```

See the syntax of the EXTEND statement for the meaning of the syntax entries not defined above.

6.11 An example

Here is a small calculator of expressions. They are given as parameters of the command.

File "calc.ml":

```
#load "pa_extend.cmo";

value g = Grammar.gcreate (Plexer.gmake ());
value e = Grammar.Entry.create g "expression";

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

open Printf;

for i = 1 to Array.length Sys.argv - 1 do {
  let r = Grammar.Entry.parse e (Stream.of_string Sys.argv.(i)) in
  printf "%s = %d\n" Sys.argv.(i) r;
  flush stdout;
};
```

The link needs the library "gramlib.cma" provided with Camlp5:

```
ocamlc -pp camlp5r -I +camlp5 gramlib.cma test/calc.ml -o calc
```

Examples:

```
$ ./calc '239*4649'
239*4649 = 1111111
$ ./calc '(47+2)/3'
(47+2)/3 = 16
```

Part II

Printing tools

Chapter 7

Extensible printers

This chapter describes the syntax and semantics of the extensible printers of Camlp5.

Symmetric to the extensible grammars, the extensible printers allow to define and extend printers of data or programs. A specific statement ”`EXTEND_PRINTER`” allow to define these extensions.

7.1 Getting started

A printer is a value of type ”`Eprinter.t a`” where ”`a`” is the type of the item to be printed. When applied, a printer return a string, representing the printed item.

To create a printer, one has to use the function ”`Eprinter.make`” with, as parameter, the name of the printer, (used in error messages). A printer is created empty, i.e. it fails if it is applied.

Like grammar entries, printers may have several levels. When the function ”`Eprinter.apply`” is applied to a printer, the first level is called. The function ”`Eprinter.apply_level`” allows to call a printer at some specific level possibly different from the first one. When a level does not match any value of the printed item, the next level is tested. If there is no more levels, the printer fails.

In semantic actions of printers, functions are provided to recursively call the current level and the next level. Moreover, a *printing context* variable is also given, giving the current indentation, what has to be printed before in the same line and what has to be printed after in the same line (it is not mandatory to use them).

The extension of printers can ben done with the ”`EXTEND_PRINTER`” statement added by the *parsing kit* ”`pa_extprint.cmo`”.

7.2 Syntax of the EXTEND_PRINTER statement

```
       expression ::= extend-statement
 extend-statement ::= "EXTEND_PRINTER" extend-body "END"
      extend-body ::= extend-printers
  extend-printers ::= extend-printer extend-printers
                    | <nothing>
   extend-printer ::= printer-name ":" position-opt "[" levels "]"
     position-opt ::= "FIRST"
                    | "LAST"
                    | "BEFORE" label
```

```
               | "AFTER" label
               | "LEVEL" label
               | <nothing>
       levels ::= level "|" levels
               | level
        level ::= label-opt "[" rules "]"
    label-opt ::= label
               | <nothing>
        rules ::= rule "|" rules
               | rule
         rule ::= pattern "->" expression
 printer-name ::= qualid
       qualid ::= qualid "." qualid
               | uident
               | lident
       uident ::= 'A'-'Z' ident
       lident ::= ('a'-'z' | '_' | utf8-byte) ident
        ident ::= ident-char*
   ident-char ::= ('a'-'a' | 'A'-'Z' | '0'-'9' | '_' | ''' | utf8-byte)
    utf8-byte ::= '\128'-'\255'
```

7.3 Semantics of EXTEND_PRINTER

7.3.1 Printers definition list

All printers are extended according to their corresponding definitions which start with an optional "position" and follow with the "levels" definition.

Optional position

After the colon, it is possible to specify a where to insert the defined levels:

- The identifier "`FIRST`" (resp. "`LAST`") indicates that the level must be inserted before (resp. after) all possibly existing levels of the entry. They become their first (resp. last) levels.
- The identifier "`BEFORE`" (resp. "`AFTER`") followed by a level label (a string) indicates that the levels must be inserted before (resp. after) that level, if it exists. If it does not exist, the extend statement fails at run time.
- The identifier "`LEVEL`" followed by a level label indicates that the first level defined in the extend statement must be inserted at the given level, extending and modifying it. The other levels defined in the statement are inserted after this level, and before the possible levels following this level. If there is no level with this label, the extend statement fails at run time.
- By default, if the entry has no level, the levels defined in the statement are inserted in the entry. Otherwise the first defined level is inserted at the first level of the entry, extending or modifying it. The other levels are inserted afterwards (before the possible second level which may previously exist in the entry).

Levels

After the optional "position", the *level* list follow. The levels are separated by vertical bars, the whole list being between brackets.

A level starts with an optional label, which corresponds to its name. This label is useful to specify this level in case of future extensions, using the *position* (see previous section) or for possible direct calls to this specific level.

Rules

A level is a list of *rules* separated by vertical bars, the whole list beiing between brackets.

A rule is an usual pattern association (in a function or in the "match" statement), i.e. a pattern, an arrow and an expression. The expression is the semantic action which must be of type "`string`".

7.3.2 Rules insertion

The rules are sorted by their patterns, according to the rules of the extensible functions.

7.3.3 Semantic action

The semantic action, i.e. the expression following the right arrow in rules, contain in its environment the variables bound by the pattern and three more variables:

- The variable "`curr`" which is a function which can be called to recursively call the printer to the current level,
- The variable "`next`" which is a function which can be called to call the printer to the next level,
- The variable "`pc`" which contains the printing context (see further), of type "`pr_context`".

The variables "`curr`" and "`next`" are of type:

```
pr_context -> t -> string
```

where "`t`" is the type of the printer (i.e. the type of its patterns).

The variable "`curr`", "`next`" and "`pc`" have predefined names and can hide the possible identifiers having the same names in the pattern or in the environment of the "`EXTEND_PRINTER`" statement.

7.3.4 Printing context

All semantic actions use a printing context in the variable "`pc`". Its type is defined as:

```
type pr_context =
  { ind : int;
    bef : string;
    aft : string;
    dang : string }
;
```

The fields are:

- "`ind`" : the current indendation
- "`bef`" : what has to be printed before, in the same line
- "`aft`" : what has to be printed after, in the same line
- "`dang`" : the dangling token to know whether parentheses are necessary

Notice that the printing context variable is just a convenience for the programmer of the printer, he is not obliged to use it.

7.4 The Eprinter module

See its section in the chapter ”Librabry”.

7.5 Examples

7.5.1 Parser and Printer of expressions

This example illustrates the symmetry between parsers and printers. A simple type of expressions is defined. A parser converts a string to a value of this type, and a printer converts a value of this type to a string.

In the printer, there is no use of the ”`pc`” parameter and no use of the ”`Pretty`” module. The strings are printed in one only line.

Here is the source (file ”`foo.ml`”):

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

open Printf;

type expr =
  [ Op of string and expr and expr
  | Int of int
  | Var of string ]
;

value g = Grammar.gcreate (Plexer.gmake ());
value pa_e = Grammar.Entry.create g "expr";
value pr_e = Eprinter.make "expr";

EXTEND
  pa_e:
    [ [ x = SELF; "+"; y = SELF -> Op "+" x y
      | x = SELF; "-"; y = SELF -> Op "-" x y ]
    | [ x = SELF; "*"; y = SELF -> Op "*" x y
      | x = SELF; "/"; y = SELF -> Op "/" x y ]
    | [ x = INT -> Int (int_of_string x)
      | x = LIDENT -> Var x
      | "("; x = SELF; ")" -> x ] ]
  ;
END;

EXTEND_PRINTER
  pr_e:
    [ [ Op "+" x y -> sprintf "%s + %s" (curr pc x) (next pc y)
      | Op "-" x y -> sprintf "%s - %s" (curr pc x) (next pc y) ]
    | [ Op "*" x y -> sprintf "%s * %s" (curr pc x) (next pc y)
      | Op "/" x y -> sprintf "%s / %s" (curr pc x) (next pc y) ]
    | [ Int x -> string_of_int x
      | Var x -> x
      | x -> sprintf "(%s)" (Eprinter.apply pr_e pc x) ] ]
```

```
  ;
END;

value parse s = Grammar.Entry.parse pa_e (Stream.of_string s);
value print e = Eprinter.apply pr_e Eprinter.empty_pc e;

if Sys.interactive.val then ()
else print_endline (print (parse Sys.argv.(1)));
```

Remark the use of "curr" and "next" while printing operators. Because of left associativity, the first operand uses "curr" and the second operand uses "next". For right associativity operators, they should be inverted. For no associativity, both should use "next".

The last line of the file allows to use it either in OCaml toplevel or as standalone program, taking the string to be printed as parameter. It can be compiled this way:

```
ocamlc -pp camlp5r -I +camlp5 gramlib.cma foo.ml
```

Examples of use (notice the redundant parentheses automatically removed by the printing algorithm):

```
$ ./a.out "(3 * x) + (2 / y)"
3 * x + 2 / y
$ ./a.out "(x+y)*(x-y)"
(x + y) * (x - y)
$ ./a.out "x + y - z"
x + y - z
$ ./a.out "(x + y) - z"
x + y - z
$ ./a.out "x + (y - z)"
x + (y - z)
```

7.5.2 Printing OCaml programs

Complete examples of usage of extensible printers are the printers in syntaxes and extended syntaxes provided by Camlp5 in the pretty printing *kits*:

- `pr_r.cmo`: pretty print in revised syntax
- `pr_o.cmo`: pretty print in normal syntax
- `pr_rp.cmo`: also pretty print the parsers in revised syntax
- `pr_op.cmo`: also pretty print the parsers in normal syntax

See the chapter entitled "Printing programs".

Chapter 8

Pretty print

A pretty print system is provided in the library module Pretty. It allows to pretty print data or programs. The Pretty module contains:

- The function "horiz_vertic" to specify how data has to be printed.
- The function "sprintf" to format strings.
- The variable "line_length" which is a reference specifying the maximum lines lengths.

8.1 Module description

8.1.1 horiz_vertic

The function "horiz_vertic" takes two functions as parameters. When called, it calls its first function. If that function fails with some internal error that the function "`sprintf`" below may raise, the second function is called.

The type of "horiz_vertic" is:

```
(unit -> 'a) -> (unit -> 'a) -> 'a
```

the horizontal function

The first function is said to be the "horizontal" function. It tries to pretty print the data on a single line. In the context of this function, if the strings built by the function "sprintf" (see below) contain newlines or have lengths greater than "line_length", the function fails (with a internal exception local to the module).

the vertical function

In case of failure of the "horizontal function", the second function of "horiz_vertic", the "vertical" function, is called. In the context of that function, the "sprintf" function behaves like the normal "sprintf" function of the OCaml library module "Printf".

8.1.2 sprintf

The function "sprintf" works like its equivalent in the module "Printf" of the OCaml library, and takes the same parameters. Its difference is that if it is called in the context of the first function (the "horizontal" function) of the function "horiz_vertic" (above), all strings built by "sprintf" are checked for newlines or length greater than the maximum line length. If it is the case, the "sprintf" function fails, and the horizontal function fails also.

If "sprintf" is not in the context of the horizontal function, it behaves like the usual "sprintf" function.

8.1.3 line_length

The variable "line_length" is a reference holding the maximum line length of lines printed horizontally. Its default is 78. This can be changed by the user before using "horiz_vertic".

8.2 Example

Suppose you want to pretty print the XML code `"<li>something</li>"`. If the "something" is short, you want to see:

```
<li>something</li>
```

If the "something" has several lines, you want to see that:

```
<li>
  something
</li>
```

A possible implementation is:

```
open Pretty;
horiz_vertic
  (fun () -> sprintf "<li>something</li>")
  (fun () -> sprintf "<li>\n  something\n</li>");
```

Notice that the "`sprintf`" above is the one of the library Pretty.

Notice also that, in a program displaying XML code, this "something" may contain other XML tags, and is therefore generally the result of other pretty printing functions, and the program should rather look like:

```
horiz_vertic
  (fun () -> sprintf "<li>%s</li>" (print something))
  (fun () -> sprintf "<li>\n  %s\n</li>" (print something))
```

Parts of this "something" can be printed horizontally and other vertically using other calls to "horiz_vertic" in the user function "print" above. But it is important to remark that if they are called in the context of the first function parameter of "horiz_vertic" above, only horizontal functions are accepted: the first failing "horizontal" function triggers the failure of the horizontal pretty printing.

8.3 Programming with Pretty

8.3.1 Hints

Just start with a call to "horiz_vertic".

As its first function, use "sprintf" just to concat the strings without putting any newlines or indentations, just using e.g. spaces to separate pieces of data.

As its second function, wonder how you want your data to be cut. At the cutting point or points, add newlines. Notice that you probably need to give the current indentation string as parameter of the called functions because they need to be taken into account in the called "horizontal" functions.

In the example below, don't put the indentation in the sprintf function but give it as parameter of your "print" function:

```
horiz_vertic
  (fun () -> sprintf "<li>%s</li>" (print "" something))
  (fun () -> sprintf "<li>\n%s\n</li>" (print "  " something))
```

Now, the "print" function could look like, supposing you print other things with "other" of the current indentation and "things" with a new shifted one:

```
value print ind something =
  horiz_vertic
    (fun () -> sprintf "%sother things..." ind)
    (fun () -> sprintf "%sother\n%s  things..." ind ind);
```

Supposing than "other" and "things" are the result of two other functions "print_other" and "print_things", your program could look like:

```
value print ind (x, y) =
  horiz_vertic
    (fun () -> sprintf "%s%s %s" ind (print_other 0 x) (print_things 0 y))
    (fun () -> sprintf "%s\n%s" (print_other ind x) (print_things (ind ^ "  ") y));
```

8.3.2 How to cancel a horizontal print

If you want to prevent a pretty printing function to be called in an horizontal context, constraining the pretty print to be on several lines in the calling function, just do:

```
horiz_vertic
  (fun () -> sprintf "\n")
  (fun () -> ... (* your normal pretty print *))
```

In this case, the horizontal print always fails, due to the newline character in the sprintf format.

8.4 Remarks

8.4.1 Kernel

The module "Pretty" is supposed to be a basic, a "kernel" module to pretty print data. It supposes that the user takes care himself of the indentation. Programs using "Pretty" are not as short as the ones using "Format" of the OCaml library, but are more flexible. Later, it is planed to find a way to extend "Pretty" with functions allowing to use a short syntax similar to the "@" convention of the function "printf" of "Format", and taking care of the indentation for the user, resulting on shorter programs.

8.4.2 Strings vs Channels

In "Pretty", the pretty print is done only on strings, not on files. To pretty print on files, just built the strings and print them afterwards with the usual output functions. Notice that OCaml allocates and frees strings very fast, and if pretty printed values are not huge, which is generally the case, it is not a real problem, memory sizes these days being much more than enough for this job.

8.4.3 Strings or other types

The "horiz_vertic" function can return values of other types than "string". For example, if you are interested only in the result of horizontal context and not on the vertical one, it is perfectly correct to write:

```
horiz_vertic
  (fun () -> Some (sprintf "I hold on a single line")
  (fun () -> None)
```

8.4.4 Why raising exceptions ?

One could ask why this pretty print system has to raise internal exceptions. Why not simply write the pretty printing program like this:

1. first build the data horizontally (without newlines)
2. if the string length is lower than the maximum line length, return it
3. if not, build the string by adding newlines in the specific places

This method works but is generally very slow (exponential in time). Because while printing horizontally, many unuseful strings are built. If, for example, the final printed data holds on 50 lines, tenth of lines may be build and build again unusefully before the overflowing is tested.

Part III

Language extensions

Chapter 9

Locations

The location is a concept often used in Camlp5, bound to know where the errors occur in the source. The basic type is "`Stdpp.location`" which is an abstract type.

9.1 Definitions

A location is internally a couple of source *positions*: the beginning and the end of an element in the source (file or interactive). A located element can be a character (the end is just the beginning plus one), a token, or a longer sequence generally corresponding to a grammar rule.

A *position* is a count of characters since the beginning of the file, starting at zero. When a couple of positions define a location, the first position is the position of the first character of the element, and the last position is the first character *not* part of the element. The location length is the difference between those two numbers. Notice that the position corresponds exactly to the character count in the streams of characters.

In the extensible grammars, a variable with the specific name "`loc`" is predefined in all semantic actions: it is the location of the associated rule. Since the syntax tree quotations generate nodes with "`loc`" as location part, this allow to generate grammars without having to think about source locations.

It is possible to change the name "`loc`" into another name, through the parameter "`-loc`" of the Camlp5 commands.

Remark: the reason why the type "`location`" is abstract is that in future versions, it may contain other informations, such as the associated comments, the type (for expressions nodes), things like that, without having to change the already written programs.

9.2 Building locations

Tools are provided in the module "`Stdpp`" to manage locations.

First, "`Stdpp.dummy_loc`" is a dummy location used when the element does not correspond to any source, or if the programmer does not want to busy about locations.

The function "`Stdpp.make_lined_loc`" builds a location from three parameters:

- the line number, starting at 1

- the position of the first column of the line
- a couple of positions of the location: the first one belonging to the given line, the second one being able to belong to another line, further.

If the line number is not known, it is possible to use the function "`Stdpp.make_loc`" taking only the couple of positions of the location. In this case, error messages may indicate the first line and a big count of characters from this line (actually from the beginning of the file). With a good text editor, it is possible, anyway to find the good location, anyway.

If the location is built with "`Stdpp.make_loc`", and if your program displays a source location itself, it is possible to use the function "`Stdpp.line_of_loc`" which takes the file name and the location as parameters and return, by reading that file, the line number, and the character positions of the location.

9.3 Raising with a location

The function "`Stdpp.raise_with_loc`" allows to raise an exception together with a location. It is the case of all exceptions raised in the extensible grammars. The raised exception is "`Stdpp.Exc_located`" with two parameters: the location and the exception itself.

Notice that "`raise_with_loc`" just reraises the exception if it is already the exception "`Exc_located`", ignoring then the new given location.

A good usage to print exceptions possibly enclosed by "`Exc_located`" is to write the "`try..with`" statement like this:

```
try ... with exn ->
  let exn =
    match exn with
    [ Stdpp.Exc_located loc exn -> do { ... print the location ...; exn }
    | _ -> exn ]
  in
  match exn with
  ...print the exception which is *not* located...
```

9.4 Other functions

Some other functions are provided:

`Stdpp.first_pos`

returns the first position (an integer) of the location.

`Stdpp.last_pos`

returns the last position (an integer) of the location (position of the first character not belonging to the element.

`Stdpp.line_nb`

returns the line number of the location or `-1` if the location does not contain a line number (i.e. built by "`Stdpp.make_loc`").

`Stdpp.bol_pos`

returns the position of the beginning of the line of the location. It is zero if the location does not contain a line number (i.e. built by "`Stdpp.make_loc`").

And still other ones used in Camlp5 sources:

`Stdpp.encl_loc`

"`Stdpp.encl_loc loc1 loc2`" returns the location starting at the smallest begin of "`loc1`" and "`loc2`" and ending at their greatest end.. In simple words, it is the location enclosing "`loc1`" and "`loc2`" and all what is between them.

`Stdpp.shift_loc`

"`shift_loc sh loc`" returns the location "`loc`" shifted with "`sh`" characters. The line number is not recomputed.

`Stdpp.sub_loc`

"`Stdpp.sub_loc loc sh len`" is the location "`loc`" shifted with "`sh`" characters and with length "`len`". The previous ending position of the location is lost.

"`Stdpp.after_loc`"

"`Stdpp.after_loc loc sh len`" is the location just after "`loc`" (i.e. starting at the end position of "`loc`"), shifted with "`sh`" characters, and of length "`len`".

Chapter 10

Syntax tree

In Camlp5, one often uses syntax trees. For example, in grammars of the language (semantic actions), in pretty printing (as patterns), in optimizing syntax code (typically streams parsers). Syntax trees are mainly defined by sum types, one for each kind of tree: "`expr`" for expressions, "`patt`" for patterns, "`ctyp`" for types, "`str_item`" for structure items, and so on. Each node corresponds to a possible value of this type.

10.1 Introduction

This syntax tree is defined in the module "`MLast`" provided by Camlp5.

For example, the syntax tree of the statement "`if`" can be written:

```
MLast.ExIfe loc e1 e2 e3
```

where "`loc`" is the location in the source, and "`e1`", "`e2`" and "`e3`" are respectively the expression after the "`if`", the one after the "`then`" and the one after the "`else`".

In all programs, it is possible to manipulate syntax trees like that.

However, the quotations systems of Camlp5 provides a quotation kit named "`q_MLast.cmo`". When loaded, it is possible to represent the syntax trees in concrete syntax. The example above can be written:

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

The interest of this representation is when one must manipulate complicated syntax trees. An example taken from the Camlp5 sources is this quotation (found in a pattern):

```
<:expr<
  match try Some $f$ with [ Stream.Failure -> None ] with
  [ Some $p$ -> $e$
  | _ -> raise (Stream.Error $e2$) ]
>>
```

In abstract syntax, it should have been written:

```
MLast.ExMat _
  (MLast.ExTry _ (MLast.ExApp _ (MLast.ExUid _ "Some") f)
     [(MLast.PaAcc _ (MLast.PaUid _ "Stream") (MLast.PaUid _ "Failure"),
       None, MLast.ExUid _ "None")])
```

```
  [(MLast.PaApp _ (MLast.PaUid _ "Some") p, None, e);
   (MLast.PaAny _, None,
    MLast.ExApp _ (MLast.ExLid _ "raise")
      (MLast.ExApp _
         (MLast.ExAcc _ (MLast.ExUid _ "Stream") (MLast.ExUid _ "Error"))
         e2))]
```

Which is less readable.

Instead of thinking of "a syntax tree", the programmer has to think of "a piece of program".

10.2 Location

In all syntax tree nodes, the first parameter is the source location of the node.

10.2.1 In expressions

When a quotation is in the context of an expression, the location parameter is "`loc`" in the node and in all its possible sub-nodes. Example: if we consider the quotation:

```
<:sig_item< value foo : int -> bool >>
```

This quotation, in a context of an expression, is equivalent to:

```
MLast.SgVal loc "foo"
  (MLast.TyArr loc (MLast.TyLid loc "int") (MLast.TyLid loc "bool"));
```

This name, "`loc`", is predefined, but it is possible to change it, using the argument "`-loc`" of the Camlp5 shell commands.

Consequently, if there is no variable "`loc`" defined in the context of the quotation, or if it is not of the good type, a semantic error occur in the OCaml compiler.

Note that in the extensible grammars, the variable "`loc`" is bound, in all semantic actions, to the location of the rule.

If the created node has no location, the simplest way is to define the variable "`loc`" to have the value "`Stdpp.dummy_loc`".

10.2.2 In patterns

When a quotation is in the context of a pattern, the location parameter of all nodes and possible sub-nodes is set to the wildcard ("`_`"). The same example above:

```
<:sig_item< value foo : int -> bool >>
```

is equivalent, in a pattern, to:

```
MLast.SgVal _ "foo"
  (MLast.TyArr _ (MLast.TyLid _ "int") (MLast.TyLid _ "int"))
```

10.3 Antiquotations

The expressions or patterns between dollar ($) characters are called *antiquotations*. In opposition to quotations which has its own syntax rules, the antiquotation is an area in the syntax of the enclosing context (expression or pattern). See the chapter about quotations.

If the quotation (any quotation) is in the context of an expression, the antiquotation is an expression. It could ba a call to a function. Examples:

```
value f e el = <:expr< [$e$ :: $loop False el$] >>;
value patt_list p pl = <:patt< ( $list:[p::pl]$) >>;
```

If the quotation (any quotation) is in the context of an pattern, the antiquotation is a pattern. Any pattern is possible, including the wildcard character ("_"). Examples:

```
fun [ <:expr< $lid:op$ $_$ $_$ >> -> op ]
match p with [ <:patt< $_$ | $_$ >> -> Some p ]
```

10.4 Nodes and Quotations

This section describes all nodes defined in the module "MLast" of Camlp5 and how to write them with quotations. Notice that, inside quotations, one is not restricted to these elementary cases, but any complex value can be used, resulting on possibly complex combined nodes.

Variables names give information of their types:

- `e`, `e1`, `e2`, `e3`: expr
- `p`, `p1`, `p2`, `p3`: patt
- `t`, `t1`, `t2`, `e3`: ctyp
- `s`: string
- `b`: bool
- `me`, `me1`, `me2`: module_expr
- `mt`, `mt1`, `mt2`: module_type
- `le`: list expr
- `lp`: list patt
- `lt`: list ctyp
- `ls`: list string
- `lse`: list (string * expr)
- `lpe`: list (patt * expr)
- `lpp`: list (patt * patt)
- `lpoee`: list (patt * option expr * expr)
- `op`: option patt
- `lcstri`: list class_str_item
- `lcsigi`: list class_sig_item

10.4.1 expr

Expressions of the language.

Node	<:expr< ... >>	Comment
ExAcc loc e1 e2	$e1$. $e2$	dot
ExAnt loc e	$anti:e$	antiquotation (1)
ExApp loc e1 e2	$e1$ $e2$	application
ExAre loc e1 e2	$e1$.($e2$)	array access
ExArr loc le	[| $list:le$ |]	array
ExAsr loc e	assert e	assert
ExAss loc e1 e2	$e1$:= $e2$	assignment
ExBae loc e le	e .{ le }	big array access
ExChr loc s	$chr:s$	character constant
ExCoe loc e (Some t1) t2	($e1$: $t1$:> $t2$)	coercion
ExCoe loc e None t2	($e1$:> $t2$)	coercion
ExFlo loc s	$flo:s$	float constant
ExFor loc s e1 e2 b le	for s = $e1$ $to:b$ $e2$ do { $list:le$ }	for
ExFun loc lopee	fun [$list:lpoee$]	function (2)
ExIfe loc e1 e2 e3	if $e1$ then $e2$ else $e3$	if
ExInt loc s ""	$int:s$	integer constant
ExInt loc s "l"	$int32:s$	integer 32 bits
ExInt loc s "L"	$int64:s$	integer 64 bits
ExInt loc s "n"	$nativeint:s$	native integer
ExLab loc s None	~ s	label
ExLab loc s (Some e)	~ s : e	label
ExLaz loc e	lazy e	lazy
ExLet loc b lpe e	let $opt:b$ $list:lpe$ in e	let binding
ExLid loc s	$lid:s$	lowercase identifier
ExLmd loc s me e	let module s = me in e	let module
ExMat loc e lpoee	match e with [$list:lpoee$]	match (2)
ExNew loc ls	new $list:ls$	new
ExObj loc op lcstri	object ($opt:op$) $list:lcstri$ end	object expression
ExOlb loc s None	? s	option label
ExOlb loc s (Some e)	? s : e	option label
ExOvr loc lse	{< lse >}	override
ExRec loc lpe None	{ $list:lpe$ }	record
ExRec loc lpe (Some e)	{ (e) with $list:lpe$ }	record
ExSeq loc le	do { $list:le$ }	sequence
ExSnd loc e s	e # s	method call
ExSte loc e1 e2	$e1$.[$e2$]	string element
ExStr loc s	$str:s$	string
ExTry loc e lpoee	try e with [$list:lpoee$]	try (2)
ExTup loc le	($list:le$)	t-uple
ExTyc loc e t	(e : t)	type constraint
ExUid loc s	$uid:s$	uppercase identifier
ExVrn loc s	` s	variant
ExWhi loc e le	while e do { $list:le$ }	while

(1) Node used to specify an antiquotation area. See the chapter about quotations.

(2) The variable "lpoee" found in "function", "match" and "try" statements correspond to a list of "`(patt * option expr * expr)`" where the "`option expr`" is the "when" optionally following the pattern:

```
p -> e
```

is represented by:

```
(p, None, e)
```

and

```
p when e1 -> e
```

is represented by:

```
(p, Some e1, e)
```

10.4.2 patt

Patterns of the language.

Node	<:patt< ... >>	Comment
PaAcc loc p1 p2	$p1$. $p2$	dot
PaAli loc p1 p2	($p1$ as $p2$)	alias
PaAnt loc p	$anti:p$	antiquotation (1)
PaAny loc	_	wildcard
PaApp loc p1 p2	$p1$ $p2$	application
PaArr loc lp	[| $list:lp$ |]	array
PaChr loc s	$chr:s$	character
PaInt loc s1 s2	$int:s$	integer
PaFlo loc s	$flo:s$	float
PaLab loc s None	~ s	label
PaLab loc s (Some p)	~ s : p	label
PaLid loc s	$lid:s$	lowercase identifier
PaOlb loc s None	? s	option label
PaOlb loc s (Some (p, None))	? s : (p)	option label
PaOlb loc s (Some (p, Some e))	? s : (p = e)	option label
PaOrp loc p1 p2	$p1$ | $p2$	or
PaRng loc p1 p2	$p1$ .. $p2$	range
PaRec loc lpp None	{ $list:lpp$ }	record
PaStr loc s	$str:s$	string
PaTup loc lp	($list:lp$)	t-uple
PaTyc loc p t	(p : t)	type constraint
PaTyp loc ls	# $list:ls$	type pattern
PaUid loc s	$uid:s$	uppercase identifier
PaVrn loc s	` s	variant

(1) Node used to specify an antiquotation area. See the chapter about quotations.

10.4.3 ctyp

Type expressions of the language.

Node	`<:ctyp< ... >>`	Comment
`TyAcc loc t1 t2`	`$t1$ . $t2$`	dot
`TyAli loc t1 t2`	`$t1$ as $t2$`	alias
`TyAny loc`	`_`	wildcard
`TyApp loc t1 t2`	`$t1$ $t2$`	application
`TyArr loc t1 t2`	`$t1$ -> $t2$`	arrow
`TyCls loc ls`	`# $list:ls$`	class
`TyLab loc s t`	`~ $s$ : $t$`	label
`TyLid loc s`	`$lid:s$`	lowercase identifier
`TyMan loc t1 t2`	`$t1$ == $t2$`	manifest
`TyObj loc lst False`	`< $list:lst$ >`	object
`TyObj loc lst True`	`< $list:lst$ .. >`	object
`TyObj loc lst b`	`< $list:lst$ $opt:b$ >`	object (general)
`TyOlb loc s t`	`? $s$ : $t$`	option label
`TyPol loc ls t`	`! $list:ls$ . $t$`	polymorph
`TyQuo loc s`	`' $s$`	variable
`TyRec loc llsbt`	`{ $list:llsbt$ }`	record
`TySum loc llslt`	`[ $list:llslt$ ]`	sum
`TyTup loc lt`	`( $list:lt$ )`	t-uple
`TyUid loc s`	`$uid:s$`	uppercase identifier
`TyVrn loc lpv None`	`[ = $list:lpv$ ]`	variant
`TyVrn loc lpv (Some None)`	`[ > $list:lpv$ ]`	variant
`TyVrn loc lpv (Some (Some []))`	`[ < $list:lpv$ ]`	variant
`TyVrn loc lpv (Some (Some ls))`	`[ < $list:lpv$ > $list:ls$ ]`	variant

10.4.4 modules...

str_item

Structure items, i.e. phrases in a ".ml" file or "struct"s elements.

Node	`<:str_item< ... >>`	Comment
`StCls loc lcd`	`class $list:lcd$`	class declaration
`StClt loc lcdt`	`class type $list:lctd$`	class type declaration
`StDcl loc lstri`	`declare $list:lstri$ end`	declare
`StDir loc s None`	`# $s$`	directive
`StDir loc s (Some e)`	`# $s$ $e$`	directive
`StDir loc s oe`	`# $s$ $opt:oe$`	directive (general)
`StExc loc s lt []`	`exception $s$ of $list:lt$`	exception
`StExc loc s lt ls`	`exception $s$ of $list:lt$ = $list:ls$`	exception
`StExp loc e`	`$exp:e$`	expression
`StExt loc s t ls`	`external $s$ : $t$ = $list:ls$`	external
`StInc loc me`	`include $me$`	include
`StMod loc b lsme`	`module $opt:b$ $list:lsme$`	module
`StMty loc s mt`	`module type $s$ = $mt$`	module type

`StOpn loc ls`	`open $list:ls$`	open
`StTyp loc ltd`	`type $list:ltd$`	type declaration
`StUse loc s lstrib`	`...internal use...`	
`StVal loc b lpe`	`value $opt:b$ $list:lpe$`	value

sig_item

Signature items, i.e. phrases in a ".mli" file or "sig"s elements.

Node	`<:sig_item< ... >>`	Comment
`SgCls loc lcd`	`class $list:lcd$`	class
`SgClt loc lct`	`class type $list:lct$`	class type
`SgDcl loc lsigi`	`declare $list:lsigi$ end`	declare
`SgDir loc s None`	`# $s$`	directive
`SgDir loc s (Some e)`	`# $s$ $e$`	directive
`SgDir loc s oe`	`# $s$ $opt:oe$`	directive (general)
`SgExc loc s []`	`exception $s$`	exception
`SgExc loc s lt`	`exception $s$ of $list:lt$`	exception
`SgExt loc s t ls`	`external $s$ : $t$ = $list:ls$`	external
`SgInc loc me`	`include $me$`	include
`SgMod loc b lsmt`	`module $opt:b$ $list:lsmt$`	module
`SgMty loc s mt`	`module type $s$ = $mt$`	module type
`SgOpn loc ls`	`open $list:ls$`	open
`SgTyp loc ltd`	`type $list:ltd$`	type declaration
`SgUse loc s lstrib`	`...internal use...`	
`SgVal loc s t`	`value $s$ : $t$`	value

module_expr

Node	`<:module_expr< ... >>`	Comment
`MeAcc loc me1 me2`	`$me1$ . $me2$`	dot
`MeApp loc me1 me2`	`$me1$ $me2$`	application
`MeFun loc s mt me`	`functor ( $s$ : $mt$ ) -> $me$`	functor
`MeStr loc lstri`	`struct $list:lstri$ end`	struct
`MeTyc loc me mt`	`( $me$ : $mt$ )`	module type constraint
`MeUid loc s`	`$uid:s$`	uppercase identifier

module_type

Node	`<:module_type< ... >>`	Comment
`MtAcc loc mt1 mt2`	`$mt1$ . $mt2$`	dot
`MtApp loc mt1 mt2`	`$mt1$ $mt2$`	application
`MtFun loc s mt1 mt2`	`functor ( $s$ : $mt1$ ) -> $mt2$`	functor

`MtLid loc s`	`$lid:s$`	lowercase identifier
`MtQuo loc s`	`' $s$`	abstract
`MtSig loc lsigi`	`sig $list:lsigi$ end`	signature
`MtUid loc s`	`$uid:s$`	uppercase identifier
`MtWit loc mt lwc`	`$mt$ with $list:lwc$`	with construction

10.4.5 classes...

class_expr

Node	<:class_expr< ... >>	Comment
`CeApp loc ce e`	`$ce$ $e$`	application
`CeCon loc ls lt`	`$list:ls$ [ $list:lt$ ]`	constructor
`CeFun loc p ce`	`fun $p$ -> $ce$`	function
`CeLet loc b lpe ce`	`let $opt:b$ $list:lpe$ in $ce$`	let binding
`CeStr loc po lcstri`	`object ($opt:op$) $list:lcstri$ end`	object
`CeTyc loc ce ct`	`($ce$ : $ct$)`	class type constraint

class_type

Node	<:class_type< ... >>	Comment
`CtCon loc ls lt`	`$list:ls$ [ $list:lt$ ]`	constructor
`CtFun loc t ct`	`[ $t$ ] -> $ct$`	arrow
`CtSig loc pt None lcsigi_item`	`object $list:lcsigi$ end`	object
`CtSig loc pt (Some t) lcsigi_item`	`object ($t$) $list:lcsigi$ end`	object
`CtSig loc pt ot lcsigi_item`	`object $opt:ot$ $list:lcsigi$ end`	object (general)

class_str_item

Node	<:class_str_item< ... >>	Comment
`CrCtr loc t1 t2`	`type $t1$ = $t2$`	type constraint
`CrDcl loc lcstri`	`declare $list:lcstri$ end`	declaration list
`CrInh loc ce None`	`inherit $ce$`	inheritance
`CrInh loc ce (Some s)`	`inherit $ce$ as $s$`	inheritance
`CrInh loc ce os`	`inherit $ce$ $opt:s$`	inheritance (general)
`CrIni loc e`	`initializer $e$`	initialization
`CrMth loc s False e None`	`method $s$ = $e$`	method
`CrMth loc s False e (Some t)`	`method $s$ : $t$ = $e$`	method
`CrMth loc s True e None`	`method private $s$ = $e$`	method
`CrMth loc s True e (Some t)`	`method private $s$ : $t$ = $e$`	method
`CrMth loc s b e ot`	`method $opt:b$ $s$ $opt:ot$ = $e$`	method (general)
`CrVal loc s False e`	`value $s$ = $e$`	value
`CrVal loc s True e`	`value mutable $s$ = $e$`	value
`CrVal loc s b e`	`value $opt:b$ $s$ = $e$`	value (general)

CrVir loc s False t	method virtual s : t	virtual method
CrVir loc s True t	method virtual private s : t	virtual method
CrVir loc s b t	method virtual $opt:b$ s : t	virtual method (general)

class_sig_item

Node	<:class_sig_item< ... >>	Comment
CgCtr loc t1 t2	type $t1$ = $t2$	type constraint
CgDcl loc lcsigi	declare $list:lcsigi$ end	declare
CgInh loc ct	inherit ct	inheritance
CgMth loc s False t	method s : t	method
CgMth loc s True t	method private s : t	method
CgMth loc s b t	method $opt:b$ s : t	method (general)
CgVal loc s False t	value s : t	value
CgVal loc s True t	value mutable s : t	value
CgVal loc s b t	value $opt:b$ s : t	value (general)
CgVir loc s False t	method virtual s : t	method virtual
CgVir loc s True t	method virtual private s : t	method virtual
CgVir loc s b t	method virtual $opt:b$ s : t	method virtual (general)

10.4.6 other

with_constr

"With" possibly following a module type.

Node	<:with_const< ... >>	Comment
WcTyp loc s ltv False t	type s $list:ltv$ = t	with type
WcTyp loc s ltv True t	type s $list:ltv$ = private t	with type
WcTyp loc s ltv b t	type s $list:ltv$ = $opt:b$ t	with type (general)
WcMod loc ls me	module $list:ls$ = me	with module

poly_variant

Polymorphic variants.

Node	<:poly_variant< ... >>	Comment
PvTag s False []	` i	constructor
PvTag s True lt	` i of & $list:lt$	constructor
PvTag s b lt	` i of $opt:b$ $list:lt$	constructor (general)
PvInh t	t	type

Chapter 11

The Pcaml module

All about language parsing entries, language printing functions, quotation management at parsing time, extensible directives, extensible options, and generalities about Camlp5.

11.1 Language parsing

11.1.1 Main parsing functions

The two functions below are called when parsing an interface (.mli file) or an implementation (.ml file) to build the syntax tree; the returned list contains the phrases (signature items or structure items) and their locations; the boolean tells whether the parser has encountered a directive; in this case, since the directive may change the syntax, the parsing stops, the directive is evaluated, and this function is called again.
These functions are references, because they can be changed to use another technology than the Camlp5 extended grammars. By default, they use the grammars entries [implem] and [interf] defined below.

```
value parse_interf :
  ref (Stream.t char -> (list (MLast.sig_item * MLast.loc) * bool));
```

Function called when parsing an interface (".mli") file

```
value parse_implem :
  ref (Stream.t char -> (list (MLast.str_item * MLast.loc) * bool));
```

Function called when parsing an implementation (".ml") file

11.1.2 Grammar

```
value gram :  Grammar.g;
```

Grammar variable of the language.

11.1.3 Entries

Grammar entries which return syntax trees. These are set by the parsing kit of the current syntax, through the statement EXTEND. They are usable by other possible user syntax extensions.

```
value expr :  Grammar.Entry.e MLast.expr;
```

Expressions.

`value patt : Grammar.Entry.e MLast.patt;`

Patterns.

`value ctyp : Grammar.Entry.e MLast.ctyp;`

Types.

`value sig_item : Grammar.Entry.e MLast.sig_item;`

Signature items, i.e. items between "`sig`" and "`end`", or inside an interface (".mli") file.

`value str_item : Grammar.Entry.e MLast.str_item;`

Structure items, i.e. items between "`struct`" and "`end`", or inside an implementation (".ml") file.

`value module_type : Grammar.Entry.e MLast.module_type;`

Module types, e.g. signatures, functors, identifiers.

`value module_expr : Grammar.Entry.e MLast.module_expr;`

Module expressions, e.g. structures, functors, identifiers.

`value let_binding : Grammar.Entry.e (MLast.patt * MLast.expr);`

Specific entry for the "let binding", i.e. the association "let pattern = expression".

`value type_declaration : Grammar.Entry.e MLast.type_decl;`

Specific entry for the "type declaration", i.e. the association "type name = type-expression"

`value class_sig_item : Grammar.Entry.e MLast.class_sig_item;`

Class signature items, i.e. items of class objects types.

`value class_str_item : Grammar.Entry.e MLast.class_str_item;`

Class structure items, i.e. items of class objects.

`value class_type : Grammar.Entry.e MLast.class_type;`

Class types, e.g. object types, class types functions, identifiers.

`value class_expr : Grammar.Entry.e MLast.class_expr;`

Class expressions, e.g. objects, class functions, identifiers.

`value interf : Grammar.Entry.e (list (MLast.sig_item * MLast.loc) * bool);`

Interface, i.e. files with extension ".mli". The location is the one of the top of the tree. The boolean says whether the parsing stopped because of the presence of a directive (which potentially could change the syntax).

`value implem : Grammar.Entry.e (list (MLast.str_item * MLast.loc) * bool);`

Implementation, i.e. files with extension ".ml". Same remark about the location and the boolean.

`value top_phrase : Grammar.Entry.e (option MLast.str_item);`

Phrases of the OCaml interactive toplevel. Return "None" in case of end of file.

`value use_file : Grammar.Entry.e (list MLast.str_item * bool);`

Phrases in files included by the directive "`#use`". The boolean indicates whether the parsing stopped because of a directive (like for "`interf`" below).

11.2 Language printing

11.2.1 Main printing functions

The two function below are called when printing an interface (.mli file) of an implementation (.ml file) from the syntax tree; the list is the result of the corresponding parsing function.
These functions are references, to allow using other technologies than the Camlp5 extended printers.

```
value print_interf :
  ref (list (MLast.sig_item * MLast.loc) -> unit);
```

Function called when printing an interface (".mli") file

```
value print_implem :
  ref (list (MLast.str_item * MLast.loc) -> unit);
```

Function called when printing an implementation (".ml") file

By default, these functions fail. The printer kit "`pr_dump.cmo`" (loaded by most Camlp5 commands) set them to functions dumping the syntax tree in binary (for the OCaml compiler). The pretty printer kits, such as "`pr_r.cmo`" and "`pr_o.cmo`" set them to functions calling the predefined printers (see next section).

11.2.2 Printers

Printers taking syntax trees as parameters and returning pretty printed strings. These are set by the printing kits, through the statement EXTEND_PRINTER. They are usable by other possible user printing extensions.

`value pr_expr : Eprinter.t MLast.expr;`

Expressions.

`value pr_patt : Eprinter.t MLast.patt;`

Patterns.

`value pr_ctyp : Eprinter.t MLast.ctyp;`

Types.

`value pr_sig_item : Eprinter.t MLast.sig_item;`

Signature items, i.e. items between "`sig`" and "`end`", or inside an interface (".mli") file.

`value pr_str_item : Eprinter.t MLast.str_item;`

Structure items, i.e. items between "`struct`" and "`end`", or inside an implementation (".ml") file.

`value pr_module_type : Eprinter.t MLast.module_type;`

Module types, e.g. signatures, functors, identifiers.

`value pr_module_expr : Eprinter.t MLast.module_expr;`

Module expressions, e.g. structures, functors, identifiers.

`value pr_class_sig_item : Eprinter.t MLast.class_sig_item;`

Class signature items, i.e. items of class objects types.

`value pr_class_str_item : Eprinter.t MLast.class_str_item;`

Class structure items, i.e. items of class objects.

value pr_class_type : Eprinter.t MLast.class_type;

Class types, e.g. object types, class types functions, identifiers.

value pr_class_expr : Eprinter.t MLast.class_expr;

Class expressions, e.g. objects, class functions, identifiers.

11.3 Quotation management

value handle_expr_quotation : MLast.loc -> (string * string) -> MLast.expr;

Called in the semantic actions of the rules parsing a quotation in position of expression.

value handle_patt_quotation : MLast.loc -> (string * string) -> MLast.patt;

Called in the semantic actions of the rules parsing a quotation in position of pattern.

value quotation_dump_file : ref (option string);

"Pcaml.quotation_dump_file" optionally tells the compiler to dump the result of an expander (of kind "generating a string") if this result is syntactically incorrect. If "None" (default), this result is not dumped. If "Some fname", the result is dumped in the file "fname". The same effect can be done with the option "-QD" of Camlp5 commands.

11.4 Extensible directives and options

type directive_fun = option MLast.expr -> unit;

The type of functions called to treat a directive with its syntactic parameter. Directives act by side effect.

value add_directive : string -> directive_fun -> unit;

Add a new directive.

value find_directive : string -> directive_fun;

Find the function associated with a directive. Raises "Not_found" if the directive does not exists.

value add_option : string -> Arg.spec -> string -> unit;

Add an option to the command line of the Camlp5 command.

11.5 Equalities over syntax trees

These equalities skip the locations.

```
value eq_expr : MLast.expr -> MLast.expr -> bool;
value eq_patt : MLast.patt -> MLast.patt -> bool;
value eq_ctyp : MLast.ctyp -> MLast.ctyp -> bool;
value eq_str_item : MLast.str_item -> MLast.str_item -> bool;
value eq_sig_item : MLast.sig_item -> MLast.sig_item -> bool;
value eq_module_expr : MLast.module_expr -> MLast.module_expr -> bool;
value eq_module_type : MLast.module_type -> MLast.module_type -> bool;
value eq_class_sig_item : MLast.class_sig_item -> MLast.class_sig_item -> bool;
value eq_class_str_item : MLast.class_str_item -> MLast.class_str_item -> bool;
value eq_class_type : MLast.class_type -> MLast.class_type -> bool;
value eq_class_expr : MLast.class_expr -> MLast.class_expr -> bool;
```

11.6 Generalities

`value version : string;`

The current version of Camlp5.

`value syntax_name : ref string;`

The name of the current syntax. Set by the loaded syntax kit.

`value input_file : ref string;`

The file currently being parsed.

`value output_file : ref (option string);`

The output file, stdout if None (default).

`value no_constructors_arity : ref bool;`

True if the current syntax does not generate constructor arity, which is the case of the normal syntax, and not of the revised one. This has an impact when converting Camlp5 syntax tree into OCaml compiler syntax tree.

Chapter 12

Extensions of syntax

Camlp5 allows to extend the syntax of the OCaml language, and even change the whole syntax.

It uses for that one of its parsing tools: the extensible grammars.

To understand the whole syntax in the examples given in this chapter, it is a good thing to know this parsing tool (the extensible grammars), but we shall try to give some minimal explanations to allow the reader to follow them.

A syntax extension is an OCaml object file (ending with ".cmo" or ".cma") which is loaded inside Camlp5. The source of this file uses calls to the specific statement EXTEND applied to entries defined in the Camlp5 module "`Pcaml`".

12.1 Entries

The grammar of OCaml contains several entries, corresponding to the major notions of the language, which are modifiable this way, and even erasable. They are defined in this module "`Pcaml`".

Most important entries:

- `expr`: the expressions.
- `patt`: the patterns.
- `ctyp`: the types.
- `str_item`: the structure items, i.e. the items between "struct" and "end", and the toplevel phrases in a ".ml" file.
- `sig_item`: the signature items, i.e. the items between "sig" and "end", and the toplevel phrases in a ".mli" file.
- `module_expr`: the module expressions.
- `module_type`: the module types.

Entries of object programming:

- `class_expr`: the class expressions.

- `class_type`: the class types.
- `class_str_item`: the objects items.
- `class_sig_item`: the objects types items.

Main entries of files and interactive toplevel parsing:

- `implem`: the phrases that can be found in a ".ml" file.
- `interf`: the phrases that can be found in a ".mli" file.
- `top_phrase`: the phrases of the interactive toplevel.
- `use_file`: the phrases that can be found in a file included by the directive "use".

Extra useful entries also accessible:

- `let_binding`: the bindings "expression = pattern" found in the "let" statement.
- `type_declaration`: the bindings "name = type" found in the "type" statement.

12.2 Syntax tree quotations

A grammar rule is a list of rule symbols followed by the semantic action, i.e. the result of the rule. This result is a syntax tree, whose type is the type of the extended entry. The description of the types of the syntax tree are in the Camlp5 module "`MLast`".

However, there is a simpler way to make values of these syntax tree types: the system quotations (see chapters about quotations and syntax tree). With this system, it is possible to represent syntax tree in concrete syntax, between specific parentheses, namely "`<<`" and "`>>`", or between "`<:name<`" and "`>>`".

For example, the syntax node of the "if" statement is, normally:

```
MLast.ExIfe loc e1 e2 e3
```

where loc is the source location, and e1, e2, e3 are the expressions constituting the if statement. With quotations, it is possible to write it like this (which is stricly equivalent because this is evaluated at parse time, not execution time):

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

With quotations, it is possible to build pieces of program as complex as desired. See the chapter about syntax trees.

12.3 An example : repeat..until

A classical extension is the programmation of the "repeat" statement. The "repeat" statement is like a "while" except that the loop is executed at least one time and that the test is at the end of the loop and is inverted. The equivalent of:

```
repeat x; y; z until c
```

is:

```
do {
  x; y; z;
  while not c do { x; y; z }
}
```

or, with a loop:

```
loop () where rec loop () = do {
  x; y; z;
  if c then () else loop ()
}
```

12.3.1 The code

This syntax extension could be written like this (see the detail of syntax in the chapter about extensible grammars and the syntax tree quotations in the chapter about them):

```
#load "pa_extend.cmo";
#load "q_MLast.cmo";
open Pcaml;
EXTEND
  expr:
    [ [ "repeat"; el = LIST1 expr SEP ";"; "until"; c = expr ->
          let el = el @ [<:expr< while not $c$ do { $list:el$ } >>] in
          <:expr< do { $list:el$ } >> ] ]
  ;
END;
```

Alternatively, with the loop version:

```
#load "pa_extend.cmo";
#load "q_MLast.cmo";
open Pcaml;
EXTEND
  expr:
    [ [ "repeat"; el = LIST1 expr SEP ";"; "until"; c = expr ->
          let el = el @ [<:expr< if $c$ then () else loop () >>] in
          <:expr< loop () where rec loop () = do { $list:el$ } >> ] ]
  ;
END;
```

The first "`#load`" in the code (in both files) means that a syntax extension has been used in the file, namely the "EXTEND" statement. The second "`#load`" means that abstract tree quotations has been used, namely the "`<:expr< ... >>`".

The quotation, found in the second version:

```
<:expr< loop () where rec loop () = do { $list:el$ } >>
```

is especially interesting. Written with abstract syntax tree, it would be:

```
MLast.ExLet loc True
  [(MLast.PaLid loc "loop",
    MLast.ExFun loc [(MLast.PaUid loc "()", None, MLast.ExSeq loc el)])]
  (MLast.ExApp loc (MLast.ExLid loc "loop") (MLast.ExUid loc "()"));
```

This shows the interest of writing abstract syntax tree with quotations: it is easier to program and to understand.

12.3.2 Compilation

If the file "foo.ml" contains one of these versions, it is possible to compile it like this:

```
ocamlc -pp camlp5r -I +camlp5 -c foo.ml
```

Notice that the ocamlc option "-c" means that we are interested only in generating the object file "foo.cmo", not achieving the compilation by creating an executable. Anyway the link would not work because of usage of modules specific to Camlp5.

12.3.3 Testing

In the OCaml toplevel

```
ocaml -I +camlp5 camlp5r.cma
        Objective Caml version ...

        Camlp5 Parsing version ...

# #load "foo.cmo";
# value x = ref 42;
value x : ref int = {val=42}
# repeat
    print_int x.val; print_endline ""; x.val := x.val + 3
  until x.val > 70;
42
45
48
51
54
57
60
63
66
69
- : unit = ()
```

In a file

The code, above, used in the toplevel, can be written in a file, say "bar.ml":

```
#load "./foo.cmo";
value x = ref 42;
repeat
  print_int x.val;
  print_endline "";
  x.val := x.val + 3
until x.val > 70;
```

with a subtile difference: the loaded file must be "`./foo.cmo`" and not just "`foo.cmo`" because Camlp5 does not have, by default, the current directory in its path.

The file can be compiled like this:

```
ocamlc -pp camlp5r bar.ml
```

or in native code:

```
ocamlopt -pp camlp5r bar.ml
```

And it is possible to check the resulting program by typing:

```
camlp5r pr_r.cmo bar.ml
```

whose displayed result is:

```
#load "./foo.cmo";
value x = ref 42;
do {
  print_int x.val;
  print_endline "";
  x.val := x.val + 3;
  while not (x.val > 70) do {
    print_int x.val;
    print_endline "";
    x.val := x.val + 3
  }
};
```

See also the same example pretty printed in its original syntax, using the extendable programs printing.

Chapter 13

Extensions of printing

Camlp5 provides extensions kits to pretty print programs in revised syntax and normal syntax. Some other extensions kits also allow to rebuild the parsers, or the EXTEND statements in their initial syntax. The pretty print system is itself extensible, by adding new rules. We present here how it works in the Camlp5 sources.

The pretty print system of Camlp5 uses the library modules Pretty, an original system to format output) and Extfun, another original system of extensible functions.

This chapter is destinated to programmers who want to understand how the pretty printing of OCaml programs work in Camlp5, want to adapt, modify or debug it, or want to add their own pretty printing extensions.

13.1 Introduction

The files doing the pretty prints are located in Camlp5 sources in the directory "etc". Look at them if you are interested on creating new ones. The main ones are:

- "etc/pr_r.ml": pretty print in revised syntax.
- "etc/pr_o.ml": pretty print in normal syntax.
- "etc/pr_rp.ml": rebuilding parsers in their original revised syntax.
- "etc/pr_op.ml": rebuilding parsers in their original normal syntax.
- "etc/pr_extend.ml": rebuilding EXTEND in its original syntax.

We present here how this system work inside these files. First, the general principles. Second, more details of the implementation.

13.2 Principles

13.2.1 Using module Pretty

All functions in OCaml pretty printing take a parameter named "the printing context" (variable `pc`). It is a record holding :

- The current indendation : `pc.ind`

- What has to be printed before, in the same line : `pc.bef`
- What has to be printed after, in the same line : `pc.aft`
- The dangling token, useful in normal syntax to know whether parentheses are necessary : `pc.dang`

A typical pretty printing function calls the function `horiz_vertic` of the library module Pretty. This function takes two functions as parameter:

- The way to print the data in one only line (*horizontal* printing)
- The way to print the data in two or more lines (*vertical* printing)

Both functions catenate the strings by using the function `sprintf` of the library module `Pretty` which controls whether the printed data holds in the line or not. They generally call, recursively, other pretty printing functions with the same behaviour.

Let us see an example (fictive) of printing an OCaml application. Let us suppose we have an application expression "`e1 e2`" to pretty print where `e1` and `e2` are sub-expressions. If both expressions and their application holds on one only line, we want to see:

```
e1 e2
```

On the other hand, if they do not hold on one only line, we want to see `e2` in another line with, say, an indendation of 2 spaces:

```
e1
  e2
```

Here is a possible implementation. The function has been named `expr_app` and can call the function `expr` to print the sub-expressions `e1` and `e2`:

```
value expr_app pc e1 e2 =
  horiz_vertic
    (fun () ->
       let s1 = expr {(pc) with aft = ""} e1 in
       let s2 = expr {(pc) with bef = ""} e2 in
       sprintf "%s %s" s1 s2)
    (fun () ->
       let s1 = expr {(pc) with aft = ""} e1 in
       let s2 =
         expr
           {(pc) with
              ind = pc.ind + 2;
              bef = tab (pc.ind + 2)}
           e2
       in
       sprintf "%s\n%s" s1 s2)
;
```

The first function is the *horizontal* printing. It ends with a `sprintf` separating the printing of `e1` and `e2` by a space. The possible "before part" (`pc.bef`) and "after part" (`pc.aft`) are transmitted in the calls of the sub-functions.

The second function is the *vertical* printing. It ends with a `sprintf` separating the printing of `e1` and `e2` by a newline. The second line starts with an indendation, using the "before part" (`pc.bef`) of the second call to `expr`.

The pretty printing library function `Pretty.horiz_vertic` calls the first (*horizontal*) function, and if it fails (either because `s1` or `s2` are too long or hold newlines, or because the final string produced by `sprintf` is too long), calls the second (*vertical*) function.

Notice that the parameter `pc` contains a field `pc.bef` (what has to be printed before in the same line), which in both cases is transmitted to the printing of `e1` (since the syntax {`(pc) with aft = ""`} is a record with `pc.bef` kept). Same for the field `pc.aft` transmitted to the printing of `e2`.

13.2.2 Using EXTEND_PRINTER statement

This system is combined to the extensible printers to allow the extensibility of the pretty printing.

The code above actually looks like:

```
EXTEND_PRINTER
  pr_expr:
    [ [ <:expr< $e1$ $e2$ >> ->
          horiz_vertic
            (fun () ->
               let s1 = curr {(pc) with aft = ""} e1 in
               let s2 = next {(pc) with bef = ""} e2 in
               sprintf "%s %s" s1 s2)
            (fun () ->
               let s1 = curr {(pc) with aft = ""} e1 in
               let s2 =
                 next
                   {(pc) with
                      ind = pc.ind + 2;
                      bef = tab (pc.ind + 2)}
                   e2
               in
               sprintf "%s\n%s" s1 s2) ] ]
  ;
END;
```

The variable "`pc`" is implicit in the semantic actions of the syntax "`EXTEND_PRINTER`", as well as two other variables: "`curr`" and "`next`".

These parameters, "`curr`" and "`next`", correspond to the pretty printing of, respectively, the current level and the next level. Since the application in OCaml is left associative, the first sub-expression is printed at the same (current) level and the second one is printed at the next level. We also see a call to `next` in the last (2nd) case of the function to treat the other cases in the next level.

13.2.3 Dangling else, bar, semicolon

In normal syntax, there are cases where it is necessary to enclose expressions between parentheses (or between begin and end, which is equivalent in that syntax). Three tokens may cause problems: the "`else`", the vertical bar "`|`" and the semicolon "`;`". Here are examples where the presence of these tokens constraints the previous expression to be parenthesized. In these three examples, removing the begin..end enclosers would change the meaning of the expression because the dangling token would be included in that expression:

Dangling else:

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

Dangling bar:

```
function
  A ->
    begin match a with
      B -> c
    | D -> e
    end
| F -> g
```

Dangling semicolon:

```
if a then b
else begin
  let c = d in
  e
end;
f
```

The information is transmitted by the value `pc.dang`. In the first example above, while displaying the "`then`" part of the outer "`if`", the sub-expression is called with the value `pc.dang` set to `"else"` to inform the last sub-sub-expression that it is going to be followed by that token. When a "`if`" expression has to be displayed without "`else`" part, and that its "`pc.dang`" is "else", it has to be enclosed with spaces.

This problem does not exist in revised syntax. While pretty printing in revised syntax, the parameter `pc.dang` is not necessary and remains the empty string.

13.2.4 By level

As explained in the chapter about the extensible printers (with the EXTEND_PRINTER statement), printers contain levels. The global printer variable of expressions is named "`pr_expr`" and contain all definitions for pretty printing expressions, organized by levels, just like the parsing of expressions. The definition of "`pr_expr`" actually looks like this:

```
EXTEND_PRINTER
  pr_expr:
    [ "top"
      [ (* code for level "top" *) ]
    | "add"
      [ (* code for level "add" *) ]
    | "mul"
      [ (* code for level "mul" *) ]
    | "apply"
      [ (* code for level "apply" *) ]
    | "simple"
      [ (* code for level "add" *) ] ]
  ;
END;
```

13.3 The Prtools module

The Prtools module is defined inside Camlp5 for pretty printing kits. It provides variables and functions to treat comments, and meta-functions to treat lists (horizontally, vertically, paragraphly).

13.3.1 Comments

`value comm_bef : pr_context -> MLast.loc -> string;`

> [comm_bef pc loc] get the comment from the source just before the given location [loc]. This comment may be reindented using [pc.ind]. Returns the empty string if no comment found.

`value source : ref string;`

> The initial source string, which must be set by the pretty printing kit. Used by [comm_bef] above.

`value set_comm_min_pos : int -> unit;`

> Set the minimum position of the source where comments can be found, (to prevent possible duplication of comments).

13.3.2 Meta functions for lists

`type pr_fun 'a = pr_context -> 'a -> string;`

> Type of printer functions.

`value hlist : pr_fun 'a -> pr_fun (list 'a);`

> [hlist elem pc el] returns the horizontally pretty printed string of a list of elements; elements are separated with spaces.
> The list is displayed in one only line. If this function is called in the context of the [horiz] function of the function [horiz_vertic] of the module Printing, and if the line overflows or contains newlines, the function internally fails (the exception is catched by [horiz_vertic] for a vertical pretty print).

`value hlist2 : pr_fun 'a -> pr_fun 'a -> pr_fun (list 'a);`

> horizontal list with a different function from 2nd element on.

`value hlistl : pr_fun 'a -> pr_fun 'a -> pr_fun (list 'a);`

> horizontal list with a different function for the last element.

`value vlist : pr_fun 'a -> pr_fun (list 'a);`

> [vlist elem pc el] returns the vertically pretty printed string of a list of elements; elements are separated with newlines and indentations.

`value vlist2 : pr_fun 'a -> pr_fun 'a -> pr_fun (list 'a);`

> vertical list with different function from 2nd element on.

`value vlist3 : pr_fun ('a * bool) -> pr_fun ('a * bool) -> pr_fun (list 'a);`

> vertical list with different function from 2nd element on, the boolean value being True for the last element of the list.

`value vlistl : pr_fun 'a -> pr_fun 'a -> pr_fun (list 'a);`

> vertical list with different function for the last element.

`value plist : pr_fun 'a -> int -> pr_fun (list ('a * string));`

[plist elem sh pc el] returns the pretty printed string of a list of elements with separators. The elements are printed horizontally as far as possible. When an element does not fit on the line, a newline is added and the element is displayed in the next line with an indentation of [sh]. [elem] is the function to print elements, [el] a list of pairs (element * separator) (the last separator being ignored).

`value plistb : pr_fun 'a -> int -> pr_fun (list ('a * string));`

[plist elem sh pc el] returns the pretty printed string of the list of elements, like with [plist] but the value of [pc.bef] corresponds to an element already printed, as it were on the list. Therefore, if the first element of [el] does not fit in the line, a newline and a tabulation is added after [pc.bef].

`value plistl : pr_fun 'a -> pr_fun 'a -> int -> pr_fun (list ('a * string));`

paragraph list with a different function for the last element.

13.3.3 Miscellaneous

`value tab : int -> string;`

[tab ind] is equivalent to [String.make ind ' ']

`value flatten_sequence : MLast.expr -> option (list MLast.expr);`

[flatten_sequence e]. If [e] is an expression representing a sequence, return the list of expressions of the sequence. If some of these expressions are already sequences, they are flattened in the list. If that list contains expressions of the form let..in sequence, this sub-sequence is also flattened with the let..in applying only to the first expression of the sequence. If [e] is a let..in sequence, it works the same way. If [e] is not a sequence nor a let..in sequence, return None.

13.4 Example : repeat..until

This pretty prints the example repeat..until statement programmed in the chapter Syntax extensions (first version generating a "`while`" statement).

13.4.1 The code

The pattern generated by the "repeat" statement is recognized (sequence ending with a "`while`" whose contents is the same than the beginning of the sequence) by the function "is_repeat" and the repeat statement is pretty printed in its initial form using the function "horiz_vertic" of the Pretty module. File "`pr_repeat.ml`":

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

open Pcaml;
open Pretty;
open Prtools;

value eq_expr_list el1 el2 =
  if List.length el1 <> List.length el2 then False
  else List.for_all2 eq_expr el1 el2
;

value is_repeat el =
  match List.rev el with
```

```
  [ [<:expr< while not $e$ do { $list:el2$ } >> :: rel1] ->
      eq_expr_list (List.rev rel1) el2
  | _ -> False ]
;

value semi_after pr pc = pr {(pc) with aft = sprintf "%s;" pc.aft};

EXTEND_PRINTER
  pr_expr:
    [ [ <:expr< do { $list:el$ } >> when is_repeat el ->
          match List.rev el with
          [ [<:expr< while not $e$ do { $list:el$ } >> :: _] ->
              horiz_vertic
                (fun () ->
                   sprintf "%srepeat %s until %s%s" pc.bef
                     (hlistl (semi_after curr) curr
                        {(pc) with bef = ""; aft = ""} el)
                     (curr {(pc) with bef = ""; aft = ""} e)
                     pc.aft)
                (fun () ->
                   let s1 = sprintf "%srepeat" (tab pc.ind) in
                   let s2 =
                     vlistl (semi_after curr) curr
                       {(pc) with
                        ind = pc.ind + 2;
                        bef = tab (pc.ind + 2);
                        aft = ""}
                       el
                   in
                   let s3 =
                     sprintf "%suntil %s" (tab pc.ind)
                       (curr {(pc) with bef = ""} e)
                   in
                   sprintf "%s\n%s\n%s" s1 s2 s3)
          | _ -> assert False ] ] ]
  ;
END;
```

13.4.2 Compilation

```
ocamlc -pp camlp5r -I +camlp5 -c pr_repeat.ml
```

13.4.3 Testing

Getting the same files "foo.ml" and "bar.ml" of the repeat syntax example:

```
$ cat bar.ml
#load "./foo.cmo";
value x = ref 42;
repeat
  print_int x.val;
  print_endline "";
```

```
    x.val := x.val + 3
  until x.val > 70;

  $ camlp
```

Without the pretty printing kit:

```
$ camlp5r pr_r.cmo bar.ml
#load "./foo.cmo";
value x = ref 42;
do {
  print_int x.val;
  print_endline "";
  x.val := x.val + 3;
  while not (x.val > 70) do {
    print_int x.val;
    print_endline "";
    x.val := x.val + 3
  }
};
```

With the pretty printing kit:

```
$ camlp5r pr_r.cmo ./pr_repeat.cmo bar.ml -l 75
#load "./foo.cmo";
value x = ref 42;
repeat
  print_int x.val;
  print_endline "";
  x.val := x.val + 3
until x.val > 70;
```

Chapter 14

Quotations

The quotations are a syntax extension in Camlp5 allowing to build expressions and patterns in any syntax independant from the one of OCaml. Quotations are *expanded*, i.e. transformed, at parse time to produce normal syntax trees, like the rest of the program. Quotations *expanders* are normal OCaml functions writable by any programmer.

The aim of quotations is to use concrete syntax for manipulating abstract values, what make programs easier to write, read, modify, and understand. Their drawback is that they are isolated from the rest of the program, in opposition to syntax extensions, which are included in the language.

14.1 Introduction

A quotation is syntactically enclosed by specific quotes formed by less (`<`) and greater (`>`) signs. Namely:

- starting with either ”`<<`” or ”`<:ident<`” where ”`ident`” is the quotation name,
- ending with ”`>>`”

Examples:

```
<< \x.x x >>
<:foo< hello, world >>
<:bar< @#$%;* >>
```

The text between these particular parentheses can be any text. It may contain enclosing quotations and the characters ”`<`”, ”`>`” and ”\” can be escaped by ”\”. Notice that possible double-quote, parentheses, OCaml comments do not have necessary to balance inside them.

As far as the lexer is concerned, a quotation is just a kind of string.

14.2 Quotation expander

The quotations are treated at parse time. Each quotation name is associated with a *quotation expander*, a function transforming the content of the quotation into a syntax tree. There are actually two expanding functions, depending on the fact that the quotation is in the context of an expression or if it is in the context of a pattern.

If a quotation has no associated quotation expander, a parsing error is displayed and the compilation fails.

The quotation expander, or, rather, expanders, are functions taking the quotation string as parameter and returning a syntax tree. There is no constraint about which parsing technology is used. It can be stream parsers, extensible grammars, string scanning, ocamllex and yacc, any.

To build syntax trees, Camlp5 provides a way to easily build them:n see the chapter about them: it is possible to build abstract syntax through concrete syntax using, precisely... quotations.

14.3 Defining a quotation

14.3.1 By syntax tree

To define a quotation, it is necessary to program the quotation expanders and to, finally, end the source code with a call to:

```
Quotation.add name (Quotation.ExAst (f_expr, f_patt));
```

where "`name`" is the name of the quotation, and "`f_expr`" and "`f_patt`" the respective quotations expanders for expressions and patterns.

Then, after compilation of the source file (without linking, i.e. using option "-c" of the OCaml compiler), an object file is created (ending with ".cmo"), which can be used as syntax extension *kit* of Camlp5.

14.3.2 By string

There is, actually, another way to program the expander: an alone function which returns, not a syntax tree, but just a string which is parsed, afterwards, by the system. This function takes a boolean as first parameter telling whether the quotation is in position of expression (True) or in position of a pattern (False).

If using that way, the source file must end with:

```
Quotation.add name (Quotation.ExStr f);
```

where "`f`" is that quotation expander. The advantage of this second approach is that it is simple to understand and use. The drawback is that there is no way to specify different source locations for different parts of the quotation (what may be important in semantic error messages).

14.3.3 Default quotation

It is possible to use some quotation without its name. Use for that the variable "`Quotation.default_quotation`". For example, ending a file by:

```
Quotation.add "foo" (Quotation.ExAst (f_expr, f_patt));
Quotation.default.val := "foo";
```

allows to use the quotation "foo" without its name, i.e.:

```
<< ... >>
```

instead of:

```
<:foo< ... >>
```

If several files set the variable "`Quotation.default`", the default quotation applies to the last loaded one.

14.4 Antiquotations

A quotation obeys its own rules of lexing and parsing. Its result is a syntax tree, of type "`Pcaml.expr`" if the quotation is in the context of an expression, or "`Pcaml.patt`" if the quotation is in the context of a pattern.

But it can be interesting to insert portions of expressions or patterns of the enclosing context in its own quotations. For that, the syntax of the quotation must define a syntax for *antiquotations areas*. It can be, for example:

- A character introducing a variable: in this case the antiquotation can just be a variable.
- A couple of characters enclosing the antiquotations. For example, in the predefined syntax tree quotations, the antiquotations are enclosed with dollar ("`$`") signs.

In quotations, the locations in the resulting syntax tree are all set to the location of the quotation itself (if this resulting tree contains locations, they are overwritten with this location). Consequently, if there are semantic (typing) errors, the OCaml compiler will underline the entire quotation.

But in antiquotations, since they can be inserted in the resulting syntax tree, it is interesting to keep their initial quotations. For that, the nodes:

```
<:expr< $anti:...$ >>
<:patt< $anti:...$ >>
```

equivalent to:

```
MLast.ExAnt loc ...
MLast.PaAnt loc ...
```

are provided (see syntax tree quotations).

Let us take an example, without this node, and with this specific node.

Let us consider an elementary quotation system whose contents is just an antiquotation. This is just a school example, since the quotations brackets are not necessary, in this case. But we are going to see how the source code is underlined in errors messages.

14.4.1 Example without antiquotation node

The code for this quotation is (file "qa.ml"):

```
#load "q_MLast.cmo";
let expr s = Grammar.Entry.parse Pcaml.expr (Stream.of_string s) in
Quotation.add "a" (Quotation.ExAst (expr, fun []));
```

The quotation expander "`expr`" just takes the string parameter (the contents of the quotation), and returns the result of the expression parser of the OCaml language.

Compilation:

```
ocamlc -pp camlp5r -I +camlp5 -c qa.ml
```

Let us test it in the toplevel with a voluntary typing error:

```
$ ocaml -I +camlp5 camlp5r.cma
        Objective Caml version ...

        Camlp5 Parsing version ...

# #load "qa.cmo";
# let x = "abc" and y = 25 in <:a< x ^ y >>;
Characters 28-41:
  let x = "abc" and y = 25 in <:a< x ^ y >>;
                              ^^^^^^^^^^^^^
This expression has type int but is here used with type string
```

We observe that the full quotation is underlined, although it concerns only the variable "y".

14.4.2 Example with antiquotation node

Let us consider this second version (file "qb.ml"):

```
#load "q_MLast.cmo";
let expr s =
  let ast = Grammar.Entry.parse Pcaml.expr (Stream.of_string s) in
  let loc = Stdpp.make_lined_loc 1 0 (0, String.length s) in
  <:expr< $anti:ast$ >>
in
Quotation.add "b" (Quotation.ExAst (expr, fun []));
```

Like above, the quotation expander "expr" takes the string parameter (the contents of the quotation) and applies the expression parser of the OCaml language. But its result, instead of being returned ast it is, is enclosed with the antiquotation node. (The location built is the location of the whole string.)

Compilation:

```
ocamlc -pp camlp5r -I +camlp5 -c qb.ml
```

Now the same test gives:

```
$ ocaml -I +camlp5 camlp5r.cma
        Objective Caml version ...

        Camlp5 Parsing version ...

# #load "qb.cmo";
# let x = "abc" and y = 25 in <:b< x ^ y >>;
Characters 37-38:
  let x = "abc" and y = 25 in <:b< x ^ y >>;
                                       ^
This expression has type int but is here used with type string
```

Notice that, now, only the variable "y" is underlined.

14.4.3 In conclusion

In the resulting tree of the quotation expander:

- only portions of this tree, which are sons of the expr and patt antiquotation nodes, have the right location they have in the quotation (provided the quotation expander gives it the right location of the antiquation in the quotation),
- the other nodes inherit, as location, the location of the full quotation.

14.5 The Quotation module

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

It is the type for quotation expander kinds:

- "`Quotation.ExStr exp`" corresponds to an expander "`exp`" returning a string which is parsed by the system to create a syntax tree. Its boolean parameter tells whether the quotation is in position of an expression (True) or in position of a pattern (False). Quotations expanders created this way may work for some particular OCaml syntax, and not for another one (e.g. may work when used with revised syntax and not when used with normal syntax, and conversely).
- "`Quotation.ExAst (expr_exp, patt_exp)`" corresponds to expanders returning syntax trees, therefore not necessiting to be parsed afterwards. The function "`expr_exp`" is called when the quotation is in position of an expression, and "`patt_exp`" when the quotation is in position of a pattern. Quotation expanders created this way are independent from the enclosing syntax.

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

"`Quotation.add name exp`" adds the quotation "`name`" associated with the expander "`exp`".

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

"`Quotation.find name`" returns the quotation expander of the given name.

```
value default :  ref string;
```

The name of the default quotation : it is possible to use this quotation between "`<<`" and "`>>`" without having to specify its name.

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

Function translating quotation names; default = identity. Used in the predefined quotation "`q_phony.cmo`". See below.

14.6 Predefined quotations

14.6.1 q_MLast.cmo

This extension kit add quotations of OCaml syntax tree, allowing to use concrete syntax to represent abstract syntax. See the chapter Syntax tree.

14.6.2 q_phony.cmo

This extension kit is destinated to pretty printing and must be loaded after a language pretty printing kit (in normal or in revised syntax). It prevents the expansions of all the quotations, transforming them into variables. The pretty printing keeps then their initial form.

Moreover the macros (extension "`pa_macro.cmo`") are also displayed in their initial form, instead of expanded.

14.7 A full example: lambda terms

This example allows to represent lambda terms by a concrete syntax and to be able to combine them using antiquotations.

A lambda term is defined like this:

```
type term =
  [ Lam of string and term
  | App of term and term
  | Var of string ]
;
```

Examples:

```
value fst = Lam "x" (Lam "y" (Var "x"));
value snd = Lam "x" (Lam "y" (Var "y"));
value delta = Lam "x" (App (Var "x") (Var "x"));
value omega = App delta delta;
value comb_s =
  Lam "x"
    (Lamb "y"
       (Lamb "z"
          (App (App (Var "x") (Var "y")) (App (Var "x") (Var "z")))));
```

Since combinations of lambda term may be complicated, The idea is to represent them by quotations in concrete syntax. We want to be able to write the examples above like this:

```
value fst = << \x.\y.x >>;
value snd = << \x.\y.y >>;
value delta = << \x.x x >>
value omega = << ^delta ^delta >>;
value comb_s = << \x.\y.\z.(x y)(x z) >>;
```

which is a classic representation of lambda terms.

Notice, in the definition of "`omega`", the usage of the caret ("`^`") sign to specify antiquotations. Notice also the simplicity of the representation of the expression defining "`comb_s`".

Here is the code of the quotation expander, term.ml. The expander uses the extensible grammars. It has its own lexer (using the stream lexers) because the lexer of OCaml programs ("`Plexer.gmake ()`"), cannot recognize the backslashes alone.

14.7.1 Lexer

```
(* lexer *)

#load "pa_lex.cmo";

value rev_implode l =
  let s = String.create (List.length l) in
  loop (String.length s - 1) l where rec loop i =
    fun
    [ [c :: l] -> do { String.unsafe_set s i c; loop (i - 1) l }
    | [] -> s ]
;

module B =
  struct
    value empty = [];
    value add x l = [x :: l];
    value get = rev_implode;
  end
;

value rec ident =
  lexer
  [ "a..zA..Z0..9-_'\128..\255" ident! | ]
;

value empty _ = parser [: _ = Stream.empty :] -> [];

value rec next_tok =
  lexer
  [ "\\" -> ("BSLASH", "")
  | "^" -> ("CARET", "")
  | "a..z" ident! -> ("IDENT", $buf)
  | "(" -> ("", "(")
  | ")" -> ("", ")")
  | "." -> ("", ".")
  | empty -> ("EOS", "")
  | -> raise (Stream.Error "lexing error: bad character") ]
;

value rec skip_spaces = lexer [ " \n\r"/ skip_spaces! | ];

value record_loc loct i (bp, ep) = do {
  if i >= Array.length loct.val then do {
    let newt =
      Array.init (2 * Array.length loct.val + 1)
        (fun i ->
           if i < Array.length loct.val then loct.val.(i)
           else Stdpp.dummy_loc)
    in
    loct.val := newt;
```

```
  }
  else ();
  loct.val.(i) := Stdpp.make_loc (bp, ep)
};

value lex_func cs =
  let loct = ref [| |] in
  let ts =
    Stream.from
      (fun i -> do {
         ignore (skip_spaces $empty cs : list char);
         let bp = Stream.count cs in
         let r = next_tok $empty cs in
         let ep = Stream.count cs in
         record_loc loct i (bp, ep);
         Some r
       })
  in
  (ts, fun i -> loct.val.(i))
;

value lam_lex =
  {Token.tok_func = lex_func;
   Token.tok_using _ = (); Token.tok_removing _ = ();
   Token.tok_match = Token.default_match;
   Token.tok_text = Token.lexer_text;
   Token.tok_comm = None}
;
```

14.7.2 Parser

```
(* parser *)

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

value g = Grammar.gcreate lam_lex;
value expr_term_eos = Grammar.Entry.create g "term";
value patt_term_eos = Grammar.Entry.create g "term";

EXTEND
  GLOBAL: expr_term_eos patt_term_eos;
  expr_term_eos:
    [ [ x = expr_term; EOS -> x ] ]
  ;
  expr_term:
    [ [ BSLASH; i = IDENT; "."; t = SELF -> <:expr< Lam $str:i$ $t$ >> ]
    | [ x = SELF; y = SELF -> <:expr< App $x$ $y$ >> ]
    | [ i = IDENT -> <:expr< Var $str:i$ >>
      | CARET; r = expr_antiquot -> r
      | "("; t = SELF; ")" -> t ] ]
  ;
```

```
  expr_antiquot:
    [ [ i = IDENT ->
          let r =
            let loc = Stdpp.make_loc (0, String.length i) in
            <:expr< $lid:i$ >>
          in
          <:expr< $anti:r$ >> ] ]
  ;
  patt_term_eos:
    [ [ x = patt_term; EOS -> x ] ]
  ;
  patt_term:
    [ [ BSLASH; i = IDENT; "."; t = SELF -> <:patt< Lam $str:i$ $t$ >> ]
    | [ x = SELF; y = SELF -> <:patt< App $x$ $y$ >> ]
    | [ i = IDENT -> <:patt< Var $str:i$ >>
      | CARET; r = patt_antiquot -> r
      | "("; t = SELF; ")" -> t ] ]
  ;
  patt_antiquot:
    [ [ i = IDENT ->
          let r =
            let loc = Stdpp.make_loc (0, String.length i) in
            <:patt< $lid:i$ >>
          in
          <:patt< $anti:r$ >> ] ]
  ;
END;

value expand_expr s =
  Grammar.Entry.parse expr_term_eos (Stream.of_string s)
;
value expand_patt s =
  Grammar.Entry.parse patt_term_eos (Stream.of_string s)
;

Quotation.add "term" (Quotation.ExAst (expand_expr, expand_patt));
Quotation.default.val := "term";
```

14.7.3 Compilation and test

Compilation:

```
ocamlc -pp camlp5r -I +camlp5 -c term.ml
```

Example, in the toplevel, including a semantic error, correctly underlined, thanks to the antiquotation nodes:

```
$ ocaml -I +camlp5 camlp5r.cma
        Objective Caml version ...

        Camlp5 Parsing version ...

# #load "term.cmo";
```

```
# type term =
   [ Lam of string and term
   | App of term and term
   |   Var of string ]
  ;
type term =
  [ Lam of string and term | App of term and term | Var of string ]
# value comb_s = << \x.\y.\z.(x y)(x z) >>;
value comb_s : term =
  Lam "x"
   (Lam "y"
     (Lam "z" (App (App (Var "x") (Var "y")) (App (Var "x") (Var "z")))))
# value omega = << ^delta ^delta >>;
Characters 18-23:
  value omega = << ^delta ^delta >>;
                   ^^^^^
Unbound value delta
# value delta = << \x.x x >>;
value delta : term = Lam "x" (App (Var "x") (Var "x"))
# value omega = << ^delta ^delta >>;
value omega : term =
  App (Lam "x" (App (Var "x") (Var "x")))
    (Lam "x" (App (Var "x") (Var "x")))
```

Chapter 15

The revised syntax

The revised syntax is an alternative syntax of OCaml. It is close to the normal syntax. We present here only the differences between the two syntaxes.

Notice that there is a simple way to know how the normal syntax is written in revised syntax: write the code in a file "foo.ml" in normal syntax and type, in a shell:

```
camlp5o pr_r.cmo pr_rp.cmo foo.ml
```

And, conversely, how a file "bar.ml" written in revised syntax is displayed in normal syntax:

```
camlp5r pr_o.cmo pr_op.cmo bar.ml
```

Even simpler, without creating a file:

```
camlp5o pr_r.cmo pr_op.cmo -impl -
... type in normal syntax ...
... type control-C ...
camlp5r pr_o.cmo pr_rp.cmo -impl -
... type in revised syntax ...
... type control-C ...
```

15.1 Modules, Structure and Signature items

- Structure and signature items always end with a single semicolon which is required.
- In structures, the declaration of a value is introduced by the keyword "value", instead of "let":

OCaml	Revised
`let x = 42;;` `let x = 42 in x + 7;;`	`value x = 42;` `let x = 42 in x + 7;`

- In signatures, the declaration of a value is also introduced by the keyword "value", instead of "val":

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

- In signatures, abstract module types are represented by a quote and an (any) identifier:

OCaml	Revised
`module type MT;;`	`module type MT = 'a;`

- Functor application uses curryfication. Parentheses are not required for the parameters:

OCaml	Revised
`type t = Set.Make(M).t;;` `module M = Mod.Make (M1) (M2);;`	`type t = (Set.Make M).t;` `module M = Mod.Make M1 M2;`

- It is possible to group several declarations together either in an interface or in an implementation by enclosing them between "declare" and "end" (this is useful when using syntax extensions to generate several declarations from one). Example in an interface:

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

15.2 Expressions and Patterns

15.2.1 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 "do" after the "while" loop and the "for" loop are followed by a "{" and the loop end with "}"; it is possible to put a semicolon after the last expression:

OCaml	Revised

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

15.2.2 Tuples and Lists

- Parentheses are required in tuples:

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

- The lists are always enclosed with brackets. A list is a left bracket, followed by a list of elements separated with semicolons, optionally followed by colon-colon and an element, and ended by a right bracket. Warning: the colon-colon is not an infix but is just part of the syntactic construction.

OCaml	Revised

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

15.2.3 Records

- In record update, parentheses are required around the initial expression:

OCaml	Revised
`{e with field = a}`	`{(e) with field = a}`

- It is authorized to use function binding syntax in record field definitions:

OCaml	Revised
`{field = fun a -> e}`	`{field a = e}`

15.2.4 Irrefutable patterns

An *irrefutable pattern* is a pattern where it is syntactilly visible that it never fails. They are used in some syntactic constructions. It is either:

- A variable,
- The wildcard "_",
- The constructor "()",
- A tuple with irrefutable patterns,
- A record with irrefutable patterns,
- A type constraint with an irrefutable pattern.

Notice that this definition is only syntactic: a constructor belonging to a type having only one constructor is not considered as an irrefutable pattern (except "()").

15.2.5 Constructions with matching

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

OCaml	Revised

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

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

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

- 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:

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

- It is possible to use a construction "where", equivalent to "let", but usable only when where is only one binding. The expression:

  ```
  e1 where p = e
  ```

  is equivalent to:

  ```
  let p = e in e1
  ```

15.2.6 Mutables and Assignment

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

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

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

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

15.2.7 Miscellaneous

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

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

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

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

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

OCaml	Revised
`(+)` `(mod)`	`\+` `\mod`

- Nested "as" patterns require parenthesis:

OCaml	Revised
`function Some a as b, c ->` `  ...`	`fun [ ((Some a as b), c) ->` `  ...`

But they are not required before the right arrow:

OCaml	Revised
`function Some a as b ->` `  ...`	`fun [ Some a as b ->` `  ...`

- The operators with special characters are not automatically infix. To define infixes, use the syntax extensions.

15.3 Types and Constructors

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

OCaml	Revised

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

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

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

- Parentheses are required in tuples of types:

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

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

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

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

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

- There is a syntax difference between data constructors with several parameters and data constructors with one parameter of type 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 curryfied:

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 curryfied, since it is alone. In that case the syntax of constructor parameters is the same between the two syntaxes:

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

- The bool constructors start with an uppercase letter. The identifiers ”true” and ”false” are not keywords:

OCaml	Revised
true && false	True && False

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

OCaml	Revised
type t = {mutable x : t1};;	type t = {x : mutable t1};

- Manifest types are with ”==”:

<table>
<tr><th>OCaml</th><th>Revised</th></tr>
<tr><td>type 'a t = 'a option =
    None
  | Some of 'a</td><td>type t 'a = option 'a ==
  [None
  | Some of 'a]</td></tr>
</table>

- Polymorph types start with ”!”:

OCaml	Revised
type t = { f : 'a . 'a list }	type t = { f : ! 'a . list 'a }

15.4 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:

<table>
<tr><th>OCaml</th><th>Revised</th></tr>
<tr><td>[< '1; '2; s; '3 >]</td><td>[: `1; `2; s; `3 :]</td></tr>
</table>

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

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

- In normal syntax, the error indicator starts with a double question mark, in revised syntax with a simple question mark:

OCaml	Revised

`parser`
`  [< '1; '2 ??  "error" >] ->`
`    ...`
	`parser`
	`  [:  ‘1; ‘2 ?  "error" :]  ->`
	`    ...`

- In normal syntax, the component optimization starts with "`?!`", in revised syntax with "`!`":

OCaml	Revised

`parser`
`  [< '1; '2 ?!  >] ->`
`    ...`
	`parser`
	`  [:  ‘1; ‘2 !  :]  ->`
	`    ...`

15.5 Classes and Objects

- Object items end with a single semicolon which is required.
- Class type parameters follow the class identifier:

OCaml	Revised
`class ['a, 'b] point = ...` `class c = [int] color;;`	`class point ['a, 'b] = ...` `class c = color [int];`

- In the type of class with parameters, the type of the parameters are between brackets. Example in signature:

OCaml	Revised
`class c :  int -> point;;`	`class c :  [int] -> point;`

- The keywords "virtual" and "private" must be in this order:

OCaml	Revised

`method virtual private m :`
`  ...`
`method private virtual m :`
`  ...`
	`method virtual private m :`
	`  ...`
	`method virtual private m :`
	`  ...`

- Object variables are introduced with "value" instead of "val":

OCaml	Revised
`object val x = 3 end`	`object value x = 3; end`

- Type constraints in objects are introduced with "type" instead of "constraint":

OCaml	Revised
`object constraint 'a = int end`	`object type 'a = int; end`

15.6 Labels and Variants

- Labels in types must start with "~":

OCaml	Revised
`val x : num:int -> bool;;`	`value x : ~num:int -> bool;`

- Types whose number of variants are fixed start with "`[ =`":

<table>
<tr><td>OCaml</td><td>Revised</td></tr>
<tr><td><code>type t = [‘On | ‘Off];;</code></td><td><code>type t = [= ‘On | ‘Off];</code></td></tr>
</table>

- The "`[`" and the "`<`" in variant types must not be sticked:

<table>
<tr><td>OCaml</td><td>Revised</td></tr>
<tr><td><code>type t = [< ‘Foo | ‘Bar];;</code></td><td><code>type t = [< ‘Foo | ‘Bar];</code></td></tr>
</table>

Chapter 16

Scheme and Lisp syntaxes

It is possible to write OCaml programs with Scheme or Lisp syntax. They are close the one to the other, both using parentheses to identify and separate statements.

16.1 Common

The syntax extension kits are named "`pa_scheme.cmo`" and "`pa_lisp.cmo`". The sources (same names ending with ".ml" in the Camlp5 sources), are written in their own syntax. They are boostrapped thanks to versions written in revised syntax and to the Camlp5 pretty printing system.

In the OCaml toplevel, it is possible to use them by loading "`camlp5r.cma`" first, then "`pa_lisp.cmo`" or "`pa_scheme.cmo`" after:

```
ocaml -I +camlp5 camlp5r.cma pa_scheme.cmo
        Objective Caml version ...

        Camlp5 Parsing version ...

# (let ((x 3)) (* 3 x))
- : int = 9
# (values 3 4 5)
- : (int * int * int) = (3, 4, 5)

ocaml -I +camlp5 camlp5r.cma pa_lisp.cmo
        Objective Caml version ...

        Camlp5 Parsing version ...

# (let ((x 3)) (* 3 x))
- : int = 9
# (, 3 4 5)
- : (int * int * int) = (3, 4, 5)
```

The grammar of Scheme and Lisp are relatively simple, just reading s-expressions. The syntax tree nodes are created in the semantic actions. Because of this, these grammars are hardly extensible.

However, the syntax extension EXTEND ("pa_extend.cmo" in extensible grammars) works for them: only the semantic actions have to be written with the Scheme or Lisp syntax. The stream parsers are also implemented.

Warning: these syntaxes are incomplete, but can be completed, if Camlp5 users are insterested.

16.2 Scheme syntax

Some examples are given to show the principles:

OCaml	Scheme
`let x = 42;;`	`(define x 42)`
`let f x = 0;;`	`(define (f x) 0)`
`let rec f x y = 0;;`	`(definerec (f x y) 0)`
`let x = 42 and y = 27 in x + y;;`	`(let ((x 42) (y 27)) (+ x y))`
`let x = 42 in let y = 27 in x + y;;`	`(let* ((x 42) (y 27)) (+ x y))`
`module M = struct ... end;;`	`(module M (struct ...))`
`type 'a t = A of 'a * int | B`	`(type (t 'a) (sum (A 'a int) (B)))`
`fun x y -> x`	`(lambda (x y) x)`
`x; y; z`	`(begin x y z)`
`f x y`	`(f x y)`
`[1; 2; 3]`	`[1 2 3]`
`x :: y :: z :: t`	`[x y z :: t]`
`a, b, c`	`(values a b c)`
`match x with 'A'..'Z' -> "x"`	`(match x ((range 'A' 'Z') "x")))`
`{x = y; z = t}`	`{(x y) (z t)}`

16.3 Lisp syntax

The same examples:

OCaml	Lisp
`let x = 42;;`	`(value x 42)`
`let f x = 0;;`	`(value f (lambda x 0))`
`let rec f x y = 0;;`	`(value rec f (lambda (x y) 0))`
`let x = 42 and y = 27 in x + y;;`	`(let ((x 42) (y 27)) (+ x y))`
`let x = 42 in let y = 27 in x + y;;`	`(let* ((x 42) (y 27)) (+ x y))`
`module M = struct ... end;;`	`(module M (struct ...))`
`type 'a t = A of 'a * int | B`	`(type (t 'a) (sum (A 'a int) (B)))`
`fun x y -> x`	`(lambda (x y) x)`
`x; y; z`	`(progn x y z)`
`f x y`	`(f x y)`
`[1; 2; 3]`	`(list 1 2 3)`
`x :: y :: z :: t`	`(list x y z :: t)`
`a, b, c`	`(, a b c)`
`match x with 'A'..'Z' -> "x"`	`(match x ((range 'A' 'Z') "x")))`
`{x = y; z = t}`	`({} (x y) (z t))`

Chapter 17

Macros

Camlp5 provides a system of macros, added by the parsing kit "pa_macro.cmo". Macros are values evaluated at parsing time.

When loaded, the parsing kit extend the syntax of the language and add command options.

17.1 Added syntax

The parsing kit "pa_macro.cmo" extends the structure items (= toplevel phrases), the expressions and the patterns by the following grammar rules:

```
      str-item ::= str-macro-def
      sig-item ::= sig-macro-def
          expr ::= macro-expr
          patt ::= macro-patt
 str_macro-def ::= "DEFINE" uident
                 | "DEFINE" uident "=" expr
                 | "DEFINE" uident "(" params ")" "=" expr
                 | "UNDEF" uident
                 | "IFDEF" dexpr "THEN" st-or-mac "END"
                 | "IFDEF" dexpr "THEN" st-or-mac "ELSE" st-or-mac "END"
                 | "IFNDEF" dexpr "THEN" st-or-mac "END"
                 | "IFNDEF" dexpr "THEN" st-or-mac "ELSE" st-or-mac "END"
 sig_macro-def ::= "DEFINE" uident
                 | "UNDEF" uident
                 | "IFDEF" dexpr "THEN" sg-or-mac "END"
                 | "IFDEF" dexpr "THEN" sg-or-mac "ELSE" sg-or-mac "END"
                 | "IFNDEF" dexpr "THEN" sg-or-mac "END"
                 | "IFNDEF" dexpr "THEN" sg-or-mac "ELSE" sg-or-mac "END"
    macro-expr ::= "IFDEF" dexpr "THEN" expr "ELSE" expr "END"
                 | "IFNDEF" dexpr "THEN" expr "ELSE" expr "END"
                 | "__FILE__"
                 | "__LOCATION__"
    macro-patt ::= "IFDEF" dexpr "THEN" patt "ELSE" patt "END"
                 | "IFNDEF" dexpr "THEN" patt "ELSE" patt "END"
    st-or-mac ::= str_macro-def
                 | str-item
```

```
sg-or-mac ::= sig-macro-def
            | sig-item
    params ::= ident "," params
            | ident
     dexpr ::= dexpr "OR" dexpr
            | dexpr "AND" dexpr
            | "NOT" dexpr
            | uident
            | "(" dexpr ")"
    uident ::= 'A'-'Z' ident
     ident ::= ident-char*
ident-char ::= ('a'-'a' | 'A'-'Z' | '0'-'9' | '_' | ''' | utf8-byte)
 utf8-byte ::= '\128'-'\255'
```

When a macro has been defined, as name e.g. "NAME", the expressions and patterns are extended this way:

```
      expr ::= "NAME"
             | "NAME" "(" expr-params ")"
      patt ::= "NAME"
             | "NAME" "(" patt-params ")"
expr-params := expr "," expr-params
             | expr
patt-params := patt "," patt-params
             | patt
```

Notice that the identifiers "`DEFINE`", "`UNDEF`", "`IFDEF`", "`IFNDEF`", "`ELSE`", "`END`", "`OR`", "`AND`" and "`NOT`" are new keywords (they cannot be used as identifiers of constructors or modules.

However, the identifiers "`__FILE__`" and "`__LOCATION__`" and the new defined macro names are not new identifiers.

17.2 Added command options

The parsing kit "`pa_macro.cmo`" also add two options usable in all Camlp5 commands:

`-D uident`

Define the uident in question like would have been a DEFINE (without parameter) in the code.

`-U uident`

Undefine the uident in question like would have been a UNDEF in the code.

`-defined`

Print the defined macros and exit.

17.3 Semantics

The statement "`DEFINE`" defines a new macro with an optional value. The macro name must start with an uppercase letter.

The existence of the macro can be tested either in structure items, in expressions or in patterns using the statement "`IFDEF`". Its non-existence can be tested by "`IFNDEF`". In expressions and patterns, the "`ELSE`" part is required, not in structure items.

The expression behind the "IFDEF" or the "IFNDEF" statement may use the operators "OR", "AND" and "NOT" and contain parentheses.

Notice that in a "IFDEF" where the value is True (resp. False), the "ELSE" (resp "THEN") part does not need to be semantically correct (well typed), since it does not appear in the resulting syntax tree. Same for "IFNDEF" and for possible macros parameters which are not used in the associated expression.

If a macro is defined twice, its first version is lost.

The statement "UNDEF" removes a macro definition.

When associated with a value (only in structure items, not in signature items), the "DEFINE" macro acts like a variable (or like function call if it has parameters), except that the parameters are evaluated at parse time and can also be used also in pattern positions. Notice that this is a way to define constants by name in patterns. For example:

```
DEFINE WW1 = 1914;
DEFINE WW2 = 1939;
value war_or_year =
  fun
  [ WW1 -> "world war I"
  | WW2 -> "world war II"
  | _ -> "not a war" ]
;
```

Notice that in a definition of a macro, if the expression contains an evaluation, the evaluation is not done by Camlp5 but just transmitted as code. In this case, it does not work in pattern position. Example in the toplevel:

```
# DEFINE PLUS(x, y) = x + y;
# PLUS(3, 4);
- : int = 7
#   fun [ PLUS(3, 4) -> () ];
Toplevel input:
#   fun [ PLUS(3, 4) -> () ];
          ^^^^^^^^^^
Failure: this is not a constructor, it cannot be applied in a pattern
```

On the other hand, if the expression does not contain evaluation, this is possible:

```
# DEFINE FOO(x, y) = (x, Some y);
# FOO(True, "bar");
- : (bool * option string) = (True, Some "bar")
# fun [ FOO(_, "hello") -> 0 | _ -> 1 ];
- : ('a * option string) -> int = <fun>
```

The macro "__FILE__" is replaced by the current compiled source file name. In the OCaml toplevel, its value is the empty string.

The macro "__LOCATION__" is replaced by the the current location (two integers in number of characters from the beginning of the file, starting at zero) of the macro itself.

17.4 Predefined macros

The macro "CAMLP5" is always predefined.

The macro "OCAML_oversion" is predefined, where "oversion" is the OCaml version the Camlp5 program has been compiled with, where all characters but numbers are replaced by underscores. For example, if using OCaml 3.09.3, the macro "OCAML_3_09_3" is defined.

Moreover, for *some* Camlp5 versions (and all the versions which follows them), the macro "CAMLP5_version" is defined where "version" is the Camlp5 version where all characters but numbers are replaced by underscores. For example, in version 4.02, the macro "CAMLP5_4_02" had been defined and this macro have appeared in all versions of Camlp5 since 4.02.

To see which macros are predefined, type:

```
camlp5r pa_macro.cmo -defined
```

Chapter 18

Pragma directive

The directive "#pragma" allows to evaluate expressions at parse time, useful, for example, to test syntax extensions by the statement EXTEND without having to compile it in a separate file.

To use it, add the syntax extension "pa_pragma.cmo" in the Camlp5 command line. It add the ability to use this directive.

Example in a file, adding a syntax for the statement 'repeat' and using it just after:

```
#pragma
  EXTEND
    GLOBAL: expr;
    expr: LEVEL "top"
      [ [ "repeat"; e1 = sequence; "until"; e2 = SELF ->
            <:expr< do { $e1$; while not $e2$ do { $e1$ } } >> ] ]
    ;
    sequence:
      [ [ el = LIST1 expr_semi -> <:expr< do { $list:el$ } >> ] ]
    ;
    expr_semi:
      [ [ e = expr; ";" -> e ] ]
    ;
  END;

let i = ref 1 in
repeat print_int i.val; print_endline ""; incr i; until i.val = 10;
```

The compilation of this example (naming it "foo.ml") can be done with the command:

```
ocamlc -pp "camlp5r q_MLast.cmo pa_extend.cmo pa_pragma.cmo" -I +camlp5 foo.ml
```

Notice that it is still experimental and probably incomplete, for the moment.

Chapter 19

Extensible functions

Extensible functions allow to define functions by pattern matching which are extensible by adding of new cases which are inserted automatically at the good place by comparing the patterns. The pattern cases are ordered according to syntax trees representing them, "when" statements being inserted before the cases without "when".

Notice that extensible functions are functional: when extending a function, a new function is returned.

The extensible functions are used in the pretty printing system of Camlp5.

19.1 Syntax

The syntax of the extensible functions, when loading "pa_extfun.cmo" is the following:

```
          expression ::= extensible-function
 extensible-function ::= "extfun" expression "with" "[" match-cases "]"
         match-cases ::= match-case "|" match-cases
          match-case ::= pattern "->" expression
                       | pattern "when" expression "->" expression
```

It is actually the same syntax than the one of "match" and "try" constructions.

19.2 Semantics

The statement "extend" defined by the syntax takes an extensible function and return another extensible function with the new match cases inserted at good places into the initial extensible function.

Extensible functions are of type "`Extfun.t a b`", which is an abstract type, where "`a`" and "`b`" are respectively the type of the patterns and the type of the expressions. It corresponds to a function of type "`a -> b`".

The function "`Extfun.apply`" takes an extensible function as parameter and return the function which can be applied like a normal function.

The value "`Extfun.empty`" is an empty extensible function, of type "`Extfun.t 'a 'b`". When applied with "`Extfun.apply`" and a parameter, it raises the exception "`Extfun.Failure`" whatever the parameter.

For debugging, it is possible to use the function "`Extfun.print`" which displays the match cases of the extensible functions. Actually, only the patterns are displayed in clear, the associated expressions are not.

The match cases are inserted according to the following rules:

- The match cases are inserted in the order they are defined in the syntax "`extend`"
- A match case pattern with "when" is insered before a match case pattern without "when".
- Two match cases patterns both with "when" or both without "when" are insered according to the alphabetic order of some internal syntax tree of the patterns where bound variables names are not taken into account.
- If two match cases patterns without "when" have the same patterns internal syntax tree, the initial one is silently removed.
- If two match cases patterns with "when" have the same patterns internal syntax tree, the new one is inserted before the old one.

Part IV

Conclusion

Chapter 20

Future work

Just some ideas... All these extensions will be done if there are good basic ideas for their implementations.

20.1 extensible syntax tree quotations

A new implementation of quotation for syntax trees allowing to use the user current syntax (his extensions included) was tested some time ago, but the result was not 100satisfactory. Several implementations are possible, and compatibility with previous versions and user programs is a problem. We are continuing thinking of it.

20.2 pretty print in shorter syntax

It is also planed to improve the pretty printing system and the printers, so that the management of the lines (indentation, continuations) be handled by the system. For the moment, the programmer has to manage them using the "`pc`" variable.

20.3 extensible lexers

Extensible lexers would be a interesting extension, also. And possibly lexers using regular expressions (extensible if possible, otherwise as a different module).

20.4 utf-8

The Camlp5 lexer for OCaml programs (module Plexer) allows utf-8 characters. Since utf-8 seems to have some success among unicode formats, perhaps a reflexion to add greek characters and/or real utf-8 arrows in the syntax (in particular in types) could be interesting.

Appendix A

Commands and Files

The main command of Camlp5 is "`camlp5`". It is an OCaml program in bytecode (compiled with ocamlc, not ocamlopt), able to dynamically load OCaml object files (ending with ".cmo" and ".cma").

All other Camlp5 commands derive from that one: they are the command "`camlp5`" with some implicitely applied parameters.

All commands have an option "`-help`" which display all possible command parameters and options. Notice that some parameters (the parsing and pretting kits) may add new options. For example, the command:

```
camlp5 pr_r.cmo -help
```

prints more lines than just:

```
camlp5 -help
```

The first parameters ("load options") allow to specify parsing and printing kits (".cmo" and ".cma" files) which are loaded inside the "`camlp5`" core before any action. Other options may follow.

A.1 Parsing and Printing Kits

A.1.1 Parsing kits

language parsing kits

`pa_r.cmo`
: Revised syntax (without parsers).

`pa_rp.cmo`
: Add revised syntax parsers.

`pa_o.cmo`
: Normal syntax (without parsers). Option added:

  `-no_quot`
  : don't parse quotations, allowing to use, e.g. "`<:>`" as token.

`pa_op.cmo`
: Add normal syntax parsers.

`pa_oop.cmo`

Add normal syntax parsers without code optimization.

`pa_lex.cmo`

Add stream lexers.

parsing extension

`pa_extend.cmo`

Add the EXTEND statement. Options added:

`-split_ext`

split EXTEND by functions to turn around a PowerPC problem.

`-quotify`

generate code for quotations (internally used to synchronize q_MLast and pa_r)

`-meta_action`

undocumented (internally used for compiled version)

`pa_extend_m.cmo`

Add the specific symbols SLIST0, SLIST1, SOPT and SFLAG to the EXTEND statement.

`pa_extfold.cmo`

Add the specific symbols FOLD0 and FOLD1 to the EXTEND statement.

printing extension

`pa_extprint.cmo`

Add the EXTEND_PRINTER statement.

extensible functions

`pa_extfun.cmo`

Add the extensible function (”extfun” statement).

functional parsers

`pa_fstream.cmo`

Add the functional parsers (”fparser” statement).

other languages

`pa_lisp.cmo`

Lisp syntax.

`pa_scheme.cmo`

Scheme syntax.

`pa_sml.cmo`

SML syntax.

other parsing kits

`pa_lefteval.cmo`

Add guarantee of left evaluation in functions calls.

`pa_macro.cmo`

Add macros. Options added:

`-D <string>`
define for IFDEF statement

`-U <string>`
undefine for IFDEF statement

`-defined`
print the defined macros and exit

`pa_pragma.cmo`

Add pragma directive: evaluations at parse time

A.1.2 Printing kits

language printing kits

`pr_r.cmo`

Display in revised syntax. Added options:

`-flag <str>`
Change pretty printing behaviour according to "`<str>`":
A/a enable/disable all flags
D/d enable/disable allowing expanding 'declare'
L/l enable/disable allowing printing 'let..in' horizontally
S/s enable/disable printing sequences beginners at end of lines
default setting is "aS".

`-wflag <str>`
Change displaying 'where' statements instead of 'let':
A/a enable/disable all flags
I/i enable/disable 'where' after 'in'
L/l enable/disable 'where' after 'let..='
M/m enable/disable 'where' after 'match' and 'try'
P/p enable/disable 'where' after left parenthesis
R/r enable/disable 'where' after 'record_field..='
S/s enable/disable 'where' in sequences
T/t enable/disable 'where' after 'then' or 'else'
V/v enable/disable 'where' after 'value..='
W/w enable/disable 'where' after '`->`'
default setting is "Ars".

`-l <length>`
Maximum line length for pretty printing (default 78)

`-sep_src`
Read source file for text between phrases (default).

-sep <string>

Use this string between phrases instead of reading source.

pr_ro.cmo

Add display objects, labels and variants in revised syntax.

pr_rp.cmo

Add display parsers with their (revised) syntax.

pr_o.cmo

Display in normal syntax. Added options:

-flag <str>

Change pretty printing behaviour according to <str>:
A/a enable/disable all flags
L/l enable/disable allowing printing 'let..in' horizontally
M/m enable/disable printing double semicolons
default setting is "Am".

-l <length>

Maximum line length for pretty printing (default 78)

-sep_src

Read source file for text between phrases (default).

-sep <string>

Use this string between phrases instead of reading source.

pr_op.cmo

Add displaying parsers with their (normal) syntax.

extensible functions

pr_extfun.cmo

Add displaying extensible functions ("extfun" statement) in their initial syntax.

other language

pr_scheme.cmo

Display in Scheme syntax. Option added:

-l <length>

Maximum line length for pretty printing (default 78)

-sep <string>

Use this string between phrases instead of reading source.

pr_schemep.cmo

Add display parsers with their (Scheme) syntax.

other printing kits

`pr_extend.cmo`

Add the displaying of EXTEND statements in their initial syntax.Option added:

`-no_slist`

Don't reconstruct SLIST, SOPT, SFLAG

`pr_depend.cmo`

Display dependencies. Option added:

`-I dir`

Add "dir" to the list of search directories.

`pr_dump.cmo`

Dump the syntax tree in binary (for the OCaml compiler)

`pr_null.cmo`

No output.

A.1.3 Quotations expanders

`q_MLast.cmo`

Syntax tree quotations. Define the quotations named: "expr", "patt", "ctyp", "str_item", "sig_item", "module_type", "module_expr", "class_type", "class_expr", "class_sig_item", "class_str_item", "with_constr" and "poly_variant".

`q_phony.cmo`

Transform quotations into phony variables to be able to pretty print the quotations in their initial form (not suitable for compilation)

A.2 Commands

`camlp5r`

Shortcut for "`camlp5 pa_r.cmo pa_rp.cmo pr_dump.cmo`"

`camlp5r.opt`

Same as previous, but in native code instead of bytecode, therefore faster. But not extensible: it is not possible to add other parsing or printing kits neither in command arguments nor with the "load" directive inside sources. Suitable for compiling sources not using other syntax extensions.

`camlp5o`

Shortcut for "`camlp5 pa_o.cmo pa_op.cmo pr_dump.cmo`"

`camlp5o.opt`

Same as previous, and like "`camlp5r.opt`", faster and not extensible. Moreover, this has been produced by compilation of Camlp5 grammars, resulting in a still faster executable.

`camlp5sch`

Shortcut for "`camlp5 pa_scheme.cmo pr_dump.cmo`"

A.3 OCaml toplevel files

These object files can be loaded in the OCaml toplevel to make Camlp5 parse the input. It is possible to load them either by putting them as parameter of the toplevel, or by using the directive "load". The option "`-I +camlp5`" has to be added to the "`ocaml`" command (the OCaml toplevel).

`camlp5r.cma`

Read phrases and display results in revised syntax

`camlp5o.cma`

Read phrases and display results in normal syntax

`camlp5sch.cma`

Read phrases in Scheme syntax

A.4 Library files

The Camlp5 library is named "`gramlib.cma`" and its native code version is "`gramlib.cmxa`". They contain the modules:

- Stdpp : building and combining locations
- Token : lexing for Camlp5 grammars
- Plexer : lexer used in revised and normal syntax
- Gramext : implementation of extensible grammars
- Grammar : extensible grammars
- Extfold : functions for grammar extensions FOLD0 and FOLD1
- Extfun : functions for extensible functions
- Eprinter : extensible printers
- Fstream : functional streams
- Pretty : pretty printing on strings

This is a pure library : when linking with it, the Camlp5 program is *not* included.

Appendix B

Library

All modules defined in "gramlib.cma", but *not* including all Camlp5 modules used by the Camlp5 commands and kits.

B.1 Stdpp module

Building and combining locations (this module should be renamed "`Ploc`" one day).

`type location = 'abstract;`

B.1.1 located exceptions

`exception Exc_located of location and exn;`

> "`Exc_located loc e`" is an encapsulation of the exception "`e`" with the input location "`loc`". To be used to specify a location for an error. This exception must not be raised by "`raise`" but rather by "`raise_with_loc`" (see below), to prevent the risk of several encapsulations of "`Exc_located`".

`value raise_with_loc :  location -> exn -> 'a;`

> "`raise_with_loc loc e`", if "`e`" is already the exception "`Exc_located`", re-raise it (ignoring the new location "`loc`"), else raise the exception "`Exc_located loc e`".

B.1.2 making locations

`value make_lined_loc :  int -> int -> (int * int) -> location;`

> "`make_lined_loc line_nb bol_pos (bp, ep)`" creates a location starting at line number "`line_nb`", where the position of the beginning of the line is "`bol_pos`" and between the positions "`bp`" (included) and "`ep`" excluded. The positions are in number of characters since the begin of the stream.

`value make_loc :  (int * int) -> location;`

> "`make_loc`" is like "`make_lined_loc`" except that the line number is not provided (to be used e.g. when the line number is unknown).

`value dummy_loc :  location;`

> "`dummy_loc`" is a dummy location, used in situations when location has no meaning.

B.1.3 getting location info

`value first_pos : location -> int;`

"`first_pos loc`" returns the position of the begin of the location in number of characters since the beginning of the stream.

`value last_pos : location -> int;`

"`first_pos loc`" returns the position of the first character not of the location in number of characters since the beginning of the stream.

`value line_nb : location -> int;`

"`line_nb loc`" returns the line number of the location or "`-1`" if the location does not contain a line number (i.e. built with "`make_loc`" above).

`value bol_pos : location -> int;`

"`line_nb loc`" returns the position of the beginning of the line of the location in number of characters since the beginning of the stream, or "`0`" if the location does not contain a line number (i.e. built the with "`make_loc`" above).

B.1.4 combining locations

`value encl_loc : location -> location -> location;`

"`encl_loc loc1 loc2`" returns the location starting at the smallest start and ending at the greatest end of the locations "`loc1`" and "`loc2`". In other words, it is the location enclosing "`loc1`" and "`loc2`".

`value shift_loc : int -> location -> location;`

"`shift_loc sh loc`" returns the location "`loc`" shifted with "`sh`" characters. The line number is not recomputed.

`value sub_loc : location -> int -> int -> location;`

"`sub_loc loc sh len`" is the location "`loc`" shifted with "`sh`" characters and with length "`len`". The previous ending position of the location is lost.

`value after_loc : location -> int -> int -> location;`

"`after_loc loc sh len`" is the location just after loc (starting at the end position of "`loc`") shifted with "`sh`" characters and of length "`len`".

B.1.5 miscellaneous

`value loc_name : ref string;`

"`loc_name.val`" is the name of the location variable used in grammars and in the predefined quotations for OCaml syntax trees. Default: "`"loc"`".

`value line_of_loc : string -> location -> (string * int * int * int);`

"`line_of_loc fname loc`" reads the file "`fname`" up to the location "`loc`" and returns the real input file, the line number and the characters location in the line; the real input file can be different from "`fname`" because of possibility of line directives typically generated by /lib/cpp.

B.2 Token module

Lexing for Camlp5 grammars.

This module defines the Camlp5 lexer type to be used in extensible grammars (see module "`Grammar`"). It also provides some useful functions to create lexers (this module should be renamed "`Plexing`" one day).

```
type pattern = (string * string);
```

Type for values used by the generated code of the EXTEND statement to represent terminals in entry rules.

- The first string is the constructor name (must start with an uppercase character). When it is empty, the second string is supposed to be a keyword.
- The second string is the constructor parameter. Empty if it has no parameter (corresponding to the 'wildcard' pattern).
- The way tokens patterns are interpreted to parse tokens is done by the lexer, function "`tok_match`" below.

```
exception Error of string;
```

A lexing error exception to be used by lexers.

B.2.1 lexer type

```
type glexer 'te =
  { tok_func : lexer_func 'te;
    tok_using : pattern -> unit;
    tok_removing : pattern -> unit;
    tok_match : mutable pattern -> 'te -> string;
    tok_text : pattern -> string;
    tok_comm : mutable option (list Stdpp.location) }
```

The type for lexers compatible with Camlp5 grammars. The parameter type "`'te`" is the type of the tokens.

- The field "`tok_func`" is the main lexer function. See "`lexer_func`" type below.
- The field "`tok_using`" is a function called by the "`EXTEND`" statement to warn the lexer that a rule uses this pattern (given as parameter). This allow the lexer 1/ to check that the pattern constructor is really among its possible constructors 2/ to enter the keywords in its tables.
- The field "`tok_removing`" is a function possibly called by the "`DELETE_RULE`" statement to warn the lexer that this pattern (given as parameter) is no more used in the grammar (the grammar system maintains a number of usages of all patterns and calls this function when this number falls to zero). If it is a keyword, this allow the lexer to remove it in its tables.
- The field "`tok_match`" is a function called by the Camlp5 grammar system to ask the lexer how the input tokens have to be matched against the patterns. Warning: for efficiency, this function has to be written as a function taking patterns as parameters and, for each pattern value, returning a function matching a token, *not* as a function with two parameters.
- The field "`tok_text`" is a function called by the grammar system to get the name of the tokens for the error messages, in case of syntax error, or for the displaying of the rules of an entry.
- The field "`tok_comm`" is a mutable place where the lexer can put the locations of the comments, if its initial value is not "`None`". If it is "`None`", nothing has to be done by the lexer.

```
and lexer_func 'te = Stream.t char -> (Stream.t 'te * location_function)
```

The type of a lexer function (field "`tok_func`" of the type "`glexer`"). The character stream is the input stream to be lexed. The result is a pair of a token stream and a location function (see below) for this tokens stream.

```
and location_function = int -> Stdpp.location;
```

The type of a function giving the location of a token in the source from the token number in the stream (starting from zero).

```
value lexer_text :  pattern -> string;
```

A simple "`tok_text`" function.

```
value default_match :  pattern -> (string * string) -> string;
```

A simple "`tok_match`" function, appling to the token type "`(string * string)`".

B.2.2 lexers from parsers or ocamllex

The functions below create lexer functions either from a "`char stream`" parser or for an "`ocamllex`" function. With the returned function "`f`", it is possible to get a simple lexer (of the type "`Token.glexer`" above):

```
 { Token.tok_func = f;
   Token.tok_using = (fun _ -> ());
   Token.tok_removing = (fun _ -> ());
   Token.tok_match = Token.default_match;
   Token.tok_text = Token.lexer_text }
```

Note that a better "`tok_using`" function should check the used tokens and raise "`Token.Error`" for incorrect ones. The other functions "`tok_removing`", "`tok_match`" and "`tok_text`" may have other implementations as well.

```
value lexer_func_of_parser :
  ((Stream.t char * ref int * ref int) -> ('te * Stdpp.location)) ->
     lexer_func 'te;
```

A lexer function from a lexer written as a char stream parser returning the next token and its location. The two references with the char stream contain the current line number and the position of the beginning of the current line.

```
value lexer_func_of_ocamllex : (Lexing.lexbuf -> 'te) -> lexer_func 'te;
```

A lexer function from a lexer created by "`ocamllex`".

B.2.3 function to build a stream and a location function

```
value make_stream_and_location :
  (unit -> ('te * Stdpp.location)) -> (Stream.t 'te * location_function);
```

B.2.4 useful functions and values

`value eval_char : string -> char;`

`value eval_string : Stdpp.location -> string -> string;`

Convert a char or a string token, where the backslashes had not been interpreted into a real char or string; raise "`Failure`" if bad backslash sequence found; "`Token.eval_char (Char.escaped c)`" would returns "`c`" and "`Token.eval_string (String.escaped s)`" would return "`s`".

`value restore_lexing_info : ref (option (int * int));`

`value line_nb : ref (ref int);`

`value bol_pos : ref (ref int);`

Special variables used to reinitialize line numbers and position of beginning of line with their correct current values when a parser is called several times with the same character stream. Necessary for directives (e.g. #load or #use) which interrupt the parsing. Without usage of these variables, locations after the directives can be wrong.

B.2.5 backward compatibilities

Deprecated since version 4.08.

`type location = Stdpp.location;`

`value make_loc : (int * int) -> location;`

`value dummy_loc : location;`

B.3 Plexer module

This module contains a lexer used for OCaml syntax (revised and normal).

B.3.1 lexer

`value gmake : unit -> Token.glexer (string * string);`

"`gmake ()`" returns a lexer compatible with the extensible grammars. The returned tokens follow the normal syntax and the revised syntax lexing rules.

The token type is "`(string * string)`" just like the pattern type.

The meaning of the tokens are:

- `("", s)` is the keyword "`s`",
- `("LIDENT", s)` is the ident "`s`" starting with a lowercase letter,
- `("UIDENT", s)` is the ident "`s`" starting with an uppercase letter,
- `("INT", s)` is an integer constant whose string source is "`s`",
- `("INT_l", s)` is an 32 bits integer constant whose string source is "`s`",
- `("INT_L", s)` is an 64 bits integer constant whose string source is "`s`",
- `("INT_n", s)` is an native integer constant whose string source is "`s`",

- `("FLOAT", s)` is a float constant whose string source is "`s`",
- `("STRING", s)` is the string constant "`s`",
- `("CHAR", s)` is the character constant "`s`",
- `("TILDEIDENT", s)` is the tilde character "`~`" followed by the ident "`s`",
- `("TILDEIDENTCOLON", s)` is the tilde character "`~`" followed by the ident "`s`" and a colon "`:`",
- `("QUESTIONIDENT", s)` is the question mark "`?`" followed by the ident "`s`",
- `("QUESTIONIDENTCOLON", s)` is the question mark "`?`" followed by the ident "`s`" and a colon "`:`",
- `("QUOTATION", "t:s")` is a quotation "`t`" holding the string "`s`",
- `("ANTIQUOT", "t:s")` is an antiquotation "`t`" holding the string "`s`",
- `("EOI", "")` is the end of input.

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

Warning: the string associated with the "`STRING`" constructor is the string found in the source without any interpretation. In particular, the backslashes are not interpreted. For example, if the input is `"\n"` the string is *not* a string with one element containing the "newline" character, but a string of two elements: the backslash and the `"n"` letter.

Same thing for the string associated with the "`CHAR`" constructor.

The functions "`Token.eval_string`" and "`Token.eval_char`" allow to convert them into the real corresponding string or char value.

B.3.2 flags

`value dollar_for_antiquotation : ref bool;`

When True (default), the next call to "`Plexer.gmake ()`" returns a lexer where the dollar sign is used for antiquotations. If False, there is no antiquotations and the dollar sign can be used as normal token.

`value specific_space_dot : ref bool;`

When "`False`" (default), the next call to "`Plexer.gmake ()`" returns a lexer where there is no difference between dots which have spaces before and dots which don't have spaces before. If "`True`", dots which have spaces before return the keyword `" ."` (space dot) and the ones which don't have spaces before return the keyword `"."` (dot alone).

`value no_quotations : ref bool;`

When "`True`", all lexers built by "`Plexer.make ()`" do not lex the quotation syntax. Default is "`False`" (quotations are lexed).

B.4 Gramext module

This module is not supposed to be used by the casual user.

It shows in clear the implementations of grammars and entries types, the normal access being through the "`Grammar`" module where these types are abstract. It can be useful for programmers interested in scanning the contents of grammars and entries, for example to make analyses on them.

B.4.1 grammar type

```
type grammar 'te =
  { gtokens : Hashtbl.t Token.pattern (ref int);
    glexer : mutable Token.glexer 'te }
;
```

The visible type of grammars, i.e. the implementation of the abstract type "`Grammar.g`". It is also the implementation of an internal grammar type used in the Grammar functorial interface.

The type parameter "`'te`" is the type of the tokens, which is "`(string * string)`" for grammars built with "`Grammar.gcreate`", and any type for grammars built with the functorial interface. The field "`gtokens`" records the count of usages of each token pattern, allowing to call the lexer function "`tok_removing`" (see the Token module) when this count reaches zero. The field "`glexer`" is the lexer.

B.4.2 entry type

```
type g_entry 'te =
  { egram : grammar 'te;
    ename : string;
    elocal : bool;
    estart : mutable int -> Stream.t 'te -> Obj.t;
    econtinue : mutable int -> int -> Obj.t -> Stream.t 'te -> Obj.t;
    edesc : mutable g_desc 'te }
```

The visible type for grammar entries, i.e. the implementation of the abstract type "`Grammar.Entry.e`" and the type of entries in the Grammar functorial interface. Notice that these entry types have a type parameter which does not appear in the "`g_entry`" type (the "`'te`" parameter is, like for grammars above, the type of the tokens). This is due to the specific typing system of the EXTEND statement which sometimes has to hide real types, the OCaml normal type system not being able to type Camlp5 grammars.

Meaning of the fields:

- `egram` : the associated grammar
- `ename` : the entry name
- `elocal` : True if the entry is local (local entries are written with a star character "*" by Grammar.Entry.print)
- `estart` and `econtinue` are parsers of the entry used in the grammar machinery
- `edesc` : the entry description (see below)

```
and g_desc 'te =
  [ Dlevels of list (g_level 'te)
  | Dparser of Stream.t 'te -> Obj.t ]
```

The entry description.

- The constructor "`Dlevels`" is for entries built by "`Grammar.Entry.create`" and extendable by the EXTEND statement.
- The constructor "`Dparser`" is for entries built by "`Grammar.Entry.of_parser`".

```
and g_level 'te =
  { assoc : g_assoc;
    lname : option string;
    lsuffix : g_tree 'te;
    lprefix : g_tree 'te }
and g_assoc = [ NonA | RightA | LeftA ]
```

Description of an entry level.

- `assoc` : the level associativity
- `lname` : the level name, if any
- `lsuffix` : the tree composed of the rules starting with "`SELF`"
- `lprefix` : the tree composed of the rules not starting with "`SELF`"

```
and g_symbol 'te =
  [ Smeta of string and list (g_symbol 'te) and Obj.t
  | Snterm of g_entry 'te
  | Snterml of g_entry 'te and string
  | Slist0 of g_symbol 'te
  | Slist0sep of g_symbol 'te and g_symbol 'te
  | Slist1 of g_symbol 'te
  | Slist1sep of g_symbol 'te and g_symbol 'te
  | Sopt of g_symbol 'te
  | Sflag of g_symbol 'te
  | Sself
  | Snext
  | Stoken of Token.pattern
  | Stree of g_tree 'te ]
```

Description of a rule symbol.

- The constructor "`Smeta`" is used by the extensions FOLD0 and FOLD1
- The constructor "`Snterm`" is the representation of a non-terminal (a call to another entry)
- The constructor "`Snterml`" is the representation of a non-terminal at some given level
- The constructor "`Slist0`" is the representation of the symbol LIST0
- The constructor "`Slist0sep`" is the representation of the symbol LIST0 followed by SEP
- The constructor "`Slist1`" is the representation of the symbol LIST1
- The constructor "`Slist1sep`" is the representation of the symbol LIST1 followed by SEP
- The constructor "`Sopt`" is the representation of the symbol OPT
- The constructor "`Sflag`" is the representation of the symbol FLAG
- The constructor "`Sself`" is the representation of the symbol SELF
- The constructor "`Snext`" is the representation of the symbol NEXT
- The constructor "`Stoken`" is the representation of a token pattern
- The constructor "`Stree`" is the representation of a anonymous rule list (between brackets).

```
and g_action = Obj.t
```

The semantic action, represented by a type "`Obj.t`" because of the specific typing of the EXTEND statement (the semantic action being able to be any function type, depending on the rule).

```
and g_tree 'te =
  [ Node of g_node 'te
  | LocAct of g_action and list g_action
  | DeadEnd ]
and g_node 'te =
  { node : g_symbol 'te; son : g_tree 'te; brother : g_tree 'te }
;
```

The types of tree and tree nodes, representing a list of factorized rules in an entry level.

- The constructor "`Node`" is a representation of a symbol (field "`node`"), the rest of the rule tree (field "`son`"), and the following node, if this node fails (field "`brother`")
- The constructor "`LocAct`" is the representation of a action, which is a function having all pattern variables of the rule as parameters and returning the rule semantic action. The list of actions in the constructor correspond to possible previous actions when it happens that rules are masked by other rules.
- The constructor "`DeadEnd`" is a representation of a nodes where the tree fails or is in syntax error.

... the following lines have to be structured ...

```
type position =
  [ First
  | Last
  | Before of string
  | After of string
  | Level of string ]
;
```

The type of position where an entry extension takes place.

- `First` : corresponds to FIRST
- `Last` : corresponds to LAST
- `Before s` : corresponds to BEFORE "s"
- `After s` : corresponds to AFTER "s"
- `Level s` : corresponds to LEVEL "s"

The module contains other definitions but for internal use.

B.5 Grammar module

Extensible grammars.

This module implements the Camlp5 extensible grammars system. Grammars entries can be extended using the `EXTEND` statement, added by loading the Camlp5 "`pa_extend.cmo`" file.

B.5.1 main types and values

`type g = 'abstract;`

The type of grammars, holding entries.

```
value gcreate :  Token.glexer (string * string) -> g;
```

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

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

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

Examples:

- The call: `Grammar.tokens g ""` returns the keywords list.
- The call: `Grammar.tokens g "IDENT"` returns the list of all usages of the pattern "IDENT" in the `EXTEND` statements.

```
value glexer :  g -> Token.glexer token;
```

Return the lexer used by the grammar

```
type parsable = 'abstract;
```

```
value parsable :  g -> Stream.t char -> parsable;
```

Type and value allowing to keep the same token stream between several calls of entries of the same grammar, to prevent loss of tokens. To be used with `Entry.parse_parsable` below

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

Module to handle entries.

- `Grammar.Entry.e` : type for entries returning values of type "`'a`".
- `Grammar.Entry.create g n` : creates a new entry named "`n`" in the grammar "`g`".
- `Grammar.Entry.parse e` : returns the stream parser of the entry "`e`".
- `Grammar.Entry.parse_token e` : returns the token parser of the entry "`e`".
- `Grammar.Entry.parse_parsable e` : returns the parsable parser of the entry "`e`".
- `Grammar.Entry.name e` : returns the name of the entry "`e`".
- `Grammar.Entry.of_parser g n p` : makes an entry from a token stream parser.
- `Grammar.Entry.print e` : displays the entry "`e`" using "`Format`".
- `Grammar.Entry.find e s` : finds the entry named "`s`" in "`e`"'s rules.
- `Grammar.Entry.obj e` : converts an entry into a "`Gramext.g_entry`" allowing to see what it holds.

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

Return the grammar associated with an entry.

B.5.2 printing grammar entries

The function "`Grammar.Entry.print`" displays the current contents of an entry. Interesting for debugging, to look at the result of a syntax extension, to see the names of the levels.

The display does not include the patterns nor the semantic actions, whose sources are not recorded in the grammar entries data.

Moreover, the local entries (not specified in the GLOBAL indicator of the EXTEND statement) are indicated with a star ("`*`") to inform that they are not directly accessible.

B.5.3 clearing grammars and entries

```
module Unsafe :
  sig
    value gram_reinit : g -> Token.glexer token -> unit;
    value clear_entry : Entry.e 'a -> unit;
  end;
```

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

- `Grammar.Unsafe.reinit_gram g lex` removes the tokens of the grammar and sets "`lex`" as a new lexer for "`g`". Warning: the lexer itself is not reinitialized.
- `Grammar.Unsafe.clear_entry e` removes all rules of the entry "`e`".

B.5.4 scan entries

```
value print_entry :  Format.formatter -> Gramext.g_entry 'te -> unit;
```

General printer for all kinds of entries (obj entries).

```
value iter_entry :
  (Gramext.g_entry 'te -> unit) -> Gramext.g_entry 'te -> unit;
```

"`Grammar.iter_entry f e`" applies "`f`" to the entry "`e`" and transitively all entries called by "`e`". The order in which the entries are passed to "`f`" is the order they appear in each entry. Each entry is passed only once. *)

```
value fold_entry :  (Gramext.g_entry 'te -> 'a -> 'a) -> Gramext.g_entry 'te -> 'a -> 'a;
```

"`Grammar.fold_entry f e init`" computes "`(f eN ..  (f e2 (f e1 init)))`", where "`e1 ..  eN`" are "`e`" and transitively all entries called by "`e`". The order in which the entries are passed to "`f`" is the order they appear in each entry. Each entry is passed only once. *)

B.5.5 functorial interface

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

```
module type GLexerType =
  sig
    type te = 'x;
    value lexer : Token.glexer te;
  end;
```

The input signature for the functor "`Grammar.GMake`": "`te`" is the type of the tokens.

```
module type S =
  sig
    type te = 'x;
    type parsable = 'x;
    value parsable : Stream.t char -> parsable;
    value tokens : string -> list (string * int);
    value glexer : Token.glexer te;
    module Entry :
      sig
        type e 'a = 'y;
        value create : string -> e 'a;
        value parse : e 'a -> parsable -> 'a;
        value parse_token : e 'a -> Stream.t te -> 'a;
        value name : e 'a -> string;
        value of_parser : string -> (Stream.t te -> 'a) -> e 'a;
        value print : e 'a -> unit;
        external obj : e 'a -> Gramext.g_entry te = "%identity";
      end;
    module Unsafe :
      sig
        value gram_reinit : Token.glexer te -> unit;
        value clear_entry : Entry.e 'a -> unit;
      end;
  end;
```

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

- Grammars are not values. Functions holding a grammar as parameter do not have this parameter yet.
- The type "`parsable`" is used in function "`parse`" instead of the char stream, avoiding the possible loss of tokens.
- The type of tokens (expressions and patterns) can be any type (instead of (string * string)); the module parameter must specify a way to show them as (string * string).

```
module GMake (L : GLexerType) :  S with type te = L.te;
```

B.5.6 grammar flags

```
value error_verbose :  ref bool;
```

Flag for displaying more information in case of parsing error; default = "`False`".

```
value warning_verbose :  ref bool;
```

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

```
value strict_parsing :  ref bool;
```

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

B.6 Extfold module

Module internally used to make the symbols FOLD0 and FOLD1 work in the EXTEND statement + extension "`pa_extfold.cmo`".

B.7 Extfun module

Extensible functions.

This module implements pattern matching extensible functions which works with the parsing kit "`pa_extfun.cmo`", the syntax of an extensible function being:

```
extfun e with [ pattern_matching ]
```

See chapter : Extensible functions.

`type t 'a 'b = 'x;`

The type of the extensible functions of type `'a -> 'b`.

`value empty : t 'a 'b;`

Empty extensible function.

`value apply : t 'a 'b -> 'a -> 'b;`

Apply an extensible function.

`exception Failure;`

Match failure while applying an extensible function.

`value print : t 'a 'b -> unit;`

Print patterns in the order they are recorded in the data structure.

B.8 Eprinter module

This module allows to create printers, apply them and clear them. It is also internally used by the "`EXTEND_PRINTER`" statement.

`type t 'a = 'abstract;`

Printer type, to print values of type "`'a`".

`type pr_context = { ind : int; bef : string; aft : string; dang : string };`

Printing context.

- "`ind`" : the current indendation
- "`bef`" : what has to be printed before, in the same line
- "`aft`" : what has to be printed after, in the same line
- "`dang`" : the dangling token to know whether parentheses are necessary

`value make : string -> t 'a;`

Builds a printer. The string parameter is used in error messages. The printer is created empty and can be extended with the "`EXTEND_PRINTER`" statement.

value apply : t 'a -> pr_context -> 'a -> string;

Applies a printer, returning the printed string of the parameter.

value apply_level : t 'a -> string -> pr_context -> 'a -> string;

Applies a printer at some specific level. Raises "Failure" if the given level does not exist.

value clear : t 'a -> unit;

Clears a printer, removing all its levels and rules.

value empty_pc : pr_context;

Empty printer context, equal to {ind = 0; bef = ""; aft = ""; dang = ""}

Some other types and functions exist, for internal use.

B.9 Fstream module

This module implement functional streams.

To be used with syntax "pa_fstream.cmo". The syntax is:

- stream: "fstream [: ... :]"
- parser: "parser [[: ... :] -> ... | ...]"

Functional parsers are of type:

```
Fstream.t 'a -> option ('a * Fstream.t 'a)
```

They use limited backtrack, i.e if a rule fails, the next rule is tested with the initial stream; limited because when in case of a rule with two consecutive symbols "a" and "b", if "b" fails, the rule fails: there is no try with the next rule of "a".

type t 'a = 'x;

The type of 'a functional streams.

value from : (int -> option 'a) -> t 'a;

"Fstream.from f" returns a stream built from the function "f". To create a new stream element, the function "f" is called with the current stream count. The user function "f" must return either "Some <value>" for a value or "None" to specify the end of the stream.

value of_list : list 'a -> t 'a;

Return the stream holding the elements of the list in the same order.

value of_string : string -> t char;

Return the stream of the characters of the string parameter.

value of_channel : in_channel -> t char;

Return the stream of the characters read from the input channel.

value iter : ('a -> unit) -> t 'a -> unit;

"Fstream.iter f s" scans the whole stream s, applying function "f" in turn to each stream element encountered.

`value next : t 'a -> option ('a * t 'a);`

Return "`Some (a, s)`" where "`a`" is the first element of the stream and "`s`" the remaining stream, or "`None`" if the stream is empty.

`value empty : t 'a -> option (unit * t 'a);`

Return "`Some ((), s)`" if the stream is empty where "`s`" is itself, else "`None`".

`value count : t 'a -> int;`

Return the current count of the stream elements, i.e. the number of the stream elements discarded.

`value count_unfrozen : t 'a -> int;`

Return the number of unfrozen elements in the beginning of the stream; useful to determine the position of a parsing error (longuest path).

B.10 Pretty module

Pretty printing on strings.

`value horiz_vertic : (unit -> 'a) -> (unit -> 'a) -> 'a;`

"`horiz_vertic h v`" first calls "`h`" to print the data horizontally, i.e. without newlines. If the displaying contains newlines or if its size exceeds the maximum line length (see variable "`line_length`" below), then the function "`h`" stops and the function "`v`" is called which can print using several lines.

`value sprintf : format 'a unit string -> 'a;`

"`sprintf fmt ...`" formats some string like "`Printf.sprintf`" does, except that, if it is called in the context of the *first* function of "`horiz_vertic`" above, it checks whether the resulting string has chances to fit in the line. If not, i.e. if it contains newlines or if its length is greater than "`max_line_length.val`", the function gives up (raising some internal exception). Otherwise the built string is returned. "`sprintf`" behaves like "`Printf.sprintf`" if it is called in the context of the *second* function of "`horiz_vertic`" or without context at all.

`value line_length : ref int;`

"`line_length`" is the maximum length (in characters) of the line. Default = 78. Can be set to any other value before printing.

Appendix C

Camlp5 sources

Information for developpers of the Camlp5 program.

C.1 Kernel

The sources are composed of:

- the OCaml stuff, copied from the OCaml compiler
- the *kernel* composed of the directories:
  - odyl : the dynamic loading system
  - lib : the library
  - main : the main program camlp5
  - meta : the parsers for revised syntax, ast quotations, EXTEND statement, etc/
- the rest: directories etc, compile, ocpp

Some other directories contain configuration files, tools, documentation and manual pages.

The kernel is sufficient to make the core system work: it is possible to compile and bootstrap only it. All sources being in revised syntax, the first compilation of Camlp5 is done by a version of this kernel in pure OCaml syntax, located in the directory ocaml_src.

These sources in pure OCaml syntax are not modified by hand. When changes are done in the kernel, and when the check is done that it correctly compiles and bootstraps, the kernel in pure OCaml syntax is rebuilt using Camlp5 pretty print. This is done by the command "make bootstrap_sources".

C.2 Compatibility

This distribution of Camlp5 is compatible with several versions of OCaml. The problem is about the definition of OCaml syntax trees which may change from a version of OCaml to another. Since OCaml does not install the sources nor the compiled versions of its syntax tree, a copy of the necessary source files, borrowed from the source of the OCaml compiler has been done in the directory 'ocaml_stuff', in subdirectories with the OCaml version number.

If the present distribution of Camlp5 is not compatible with the version of OCaml you have (the command 'configure' tells you), it is possible to add it. For that, you need the sources of the OCaml distribution you have. Once done, after a 'configure' telling you that it is not compatible, do:

```
make steal OCAML_SRC=<path-to-OCaml-sources>
```

This creates a new directory in 'ocaml_stuff' with sources of the syntax tree of your OCaml compiler.

If you want to check that the sources of the syntax tree of OCaml are up-to-date (e.g. if this is the current OCaml developpement), do:

```
make compare_stolen OCAML_SRC=<path-to-OCaml-sources>
```

The compatibility is done also with the file 'main/ast2pt.ml', which is the module converting Camlp5 syntax tree into OCaml syntax tree.

In the directory 'ocaml_src' with containt the pure OCaml sources of the Camlp5 core (see chapter TREE STRUCTURE below), there are as versions of this files as version of OCaml. They are named 'ast2pt.ml_¡version¿'. If you are adding a new version of OCaml, you need this file. In a first step, make a copy from a close version:

```
cd ocaml_src/main
cp ast2pt.ml_<close_version> ast2pt.ml_<version>
```

Then, you can redo "configure" and do "make core". If the file 'ocaml_src/main/ast2pt.ml' has a compilation problems, fix them and to 'make core' again.

Later, the same file 'main/ast2pt.ml' in Camlp5 syntax may have similar compilation problem. This file is in one examplary, thanks to IFDEF used here or there.

While compiling with some specific version of OCaml, this file is compiled with 'OCAML_vers' defined where 'vers' is the version number form the beginning to the first space or charcter '+' with all dots converted into underscores. For example, if you OCaml version is 7.04.2+dev35, you can see in the compilation process of ast2pt.ml that OCAML_7_04_2 is defined, and you can add statements defined by the syntax extension 'pa_macro.cmo', for example IFDEF OCAML_7_04_2. Add statements like that in 'main/ast2pt.ml' to make it compile.

C.3 Tree structure

The directory 'ocaml_src' contains images in pure OCaml syntax of the directories odyl lib main and meta. This allow to create a core version of Camlp5 from only the OCaml compiler installed.

You can decompose the building of the Camlp5 core into:

1. make library_cold

just makes the directory 'ocaml_src/lib' and copy the cmo and cmi files into the directory 'boot'

2. make compile_cold

makes the other directories of ocaml_src

3. make promote_cold

copies the executables "camlp5", "camlp5r" and the syntax extensions (cmo files) into the directory 'boot'

From that point, the core Camlp5 is in directory 'boot'. The real sources in the top directories odyl lib main and meta, which are written in revised syntax with some syntax extensions (grammars, quotations) can be compiled. To achieve their compilation, you can do:

```
make core
```

Or to compile everything do:

```
make all
```

or just:

```
make
```

Notice that doing "make core" or "make all" from scratch (after a make clean), automatically starts by making the core files from their pure OCaml versions.

C.4 Fast compilation from scratch

```
./configure
make clean core compare
make coreboot
make all opt opt.opt
```

C.5 Testing changes

1. do your changes

2. do:

```
make core compare
```

if it says that the bootstrap is ok, you can do:

```
make all
make opt
make opt.opt
```

otherwise, to make sure everything is ok, first do:

```
make coreboot
```

sometimes two bootstraps ('make coreboot' twice) are necessary, in particular if you change things in the directory 'lib'. It is even possible that three bootstraps are necessary.

C.6 Before committing your changes

Make sure that the cold start with pure OCaml sources work. For that, do:

```
make compare_sources | less
```

This shows you the changes that would be done in the OCaml pure sources of the directory ocaml_src.

To make the new versions, do:

```
make new_sources
make promote_sources
```

Notice that these pure OCaml sources are not supposed to be modified by hand, but only created by the above commands. Besides, their sources, although pretty printed, are sometimes not easy to read, particularly for expanded grammars (of the statement 'EXTEND').

However, if these sources do not compile, due to changes in the OCaml compiler, it is possible to edit them. In this case, similar changes may have to be done in the normal sources in revised syntax.

After doing 'make new_sources' above, and before doing 'make promote_sources' below, it is possible to do 'make untouch_sources' which change the dates of the new created files with the dates of the old files if they are not modified. This way, the "svn commit" will not have to compare these files, which may have some importance if you network is not fast.

The 'make new_sources' builds a directory named 'ocaml_src.new'. If this directory still exists, due to a previous 'make new_sources', the command fails. In this case, just delete it (rm -rf ocaml_src.new) without problem: this directory is not part of the distribution, it is just temporary.

The 'make clean_sources' deletes old versions of ocaml_src, keeping only the last and the before last ones.

The command:

```
make bootstrap_sources
```

is a shortcut for:

```
make new_sources
make untouch_sources
make promote_sources
make clean_sources
```

If there are changes in the specific file 'main/ast2pt.ml', do also:

```
make compare_all_ast2pt
```

and possibly:

```
make bootstrap_all_ast2pt
```

because this file, in 'ocaml_src/main' directory has different versions according to the OCaml version.

After having rebuilt the pure OCaml sources, check they work by rebuilding everything from scratch, starting with "configure".

C.7 If you change the main parser

If you change the main parser 'meta/pa_r.ml', you may check that the quotations expanders of syntax tree 'meta/q_MLast.ml' match the new version. For that, do:

```
cd meta
make compare_q_MLast
```

If no differences are displayed, it means that 'q_MLast.ml' is ok, relatively to 'pa_r.ml'.

Otherwise, if the displayed differences seem reasonable, update the version by typing:

```
make bootstrap_q_MLast
```

Then returning to the top directory, do 'make core compare' and possibly 'make coreboot' (one of several times) to check the correctness of the file.

And don't forget, if you want to commit, to re-create the pure OCaml sources like indicated above.