Camlp4 - Reference Manual

version 3.07

Daniel de Rauglaudre
September 30, 2003

Warning: this manual may not be up-to-date with respect with the current implementation. We are planning to release a new documentation after the 3.07 final release.

Contents

Introduction

Camlp4 is a Pre-Processor-Pretty-Printer for OCaml. It offers syntactic tools (parsers, extensible grammars), the ability to extend the concrete syntax of OCaml (quotations, syntax extensions), and to redefine it from scratch.
This document holds the directions for use of Camlp4 and describes these features. It is written for programmers who already know OCaml, and are interested in concrete syntax.
Here are the contents of the present document:

- Chapter 1. "Camlp4" describes the Camlp4 features, the predefined syntaxes offered and how to use Camlp4 with the OCaml compiler and toplevel.
- Chapter 3. "Quotations" describes the Camlp4 quotation system, the first and simplest way to make syntax extensions.
- Chapter 4. "Grammars" describes the Camlp4 grammar system, a way to create grammars, more powerful and convenient than the simple stream parsers, and which are extensible what yacc and lex are not. Camlp4 predefined syntaxes use this grammar system.
- Chapter 5. "Syntax extensions" describes how to make concrete syntax extensions in OCaml and how to create a new concrete syntax for the whole language. A knowledge of the Camlp4 grammar system (chapter 4) is necessary to read this chapter.
- Chapter 6. "The revised syntax" describes an alternative concrete syntax for OCaml supposed to fix some difficulties of the standard OCaml concrete syntax. This chapter can be read directly without having read the other ones.
- Chapter 7. "Other extensions" describes the extensible functions and the functional streams.
- Chapter 8. "Camlp4 library modules" describes the interfaces of the Camlp4 library modules used by the Camlp4 tools: "`Grammar`", "`Pcaml`", "`Plexer`", "`Stdpp`", "`Token`".
- Appendix A. "Quotation for creating abstract syntax trees" describes a quotation expander provided to create language syntax trees.

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

Chapter 1

Camlp4

Camlp4 is a Pre-Processor-Pretty-Printer for OCaml, parsing a source file and printing some result. To run, Camlp4 needs to load files holding operations defining or modifying parsing (how to read the input file) and printing (how to print the result). Such files are OCaml object files (ending with "`.cmo`" or "`.cma`").
The command "`camlp4`" runs the Camlp4 program. The first options are objects ("`.cmo`") or library ("`.cma`") files which are loaded, selecting different parsing and printing operations. Among these first options, it is also possible to use the option "`-I`" to select directories where objects and library files are searched. But, important remark: by default, the system *does not* searches in the current directory: to select some object file, e.g. "`foo.cmo`" in the current directory, write "`./foo.cmo`" or add the option "`-I .`".

1.1 Camlp4 predefined files

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

- Parsing syntax:
  - `pa_o.cmo`: OCaml.
  - `pa_r.cmo`: Revised: this is an alternative syntax for the language (chapter 6).
- Printing syntax:
  - `pr_o.cmo`: Pretty print with OCaml syntax. Add an option `-l` to the command line to specify a line length.
  - `pr_r.cmo`: Pretty print with revised syntax. Add this option `-l` too.
  - `pr_dump.cmo`: Dump the syntax tree

Other predefined files allow to extend parsing and printing operations:

- Parsing extensions:
  - `pa_oop.cmo`: (to be loaded after "`pa_o.cmo`") Parsers and streams, with OCaml syntax, without code optimization, corresponding to what is internally generated by the OCaml compiler.

- pa_op.cmo: (to be loaded after "pa_o.cmo") Parsers and streams, with OCaml syntax, with code optimization.
- pa_rp.cmo: (to be loaded after "pa_r.cmo") Parsers and streams, with revised syntax and code optimization.
- pa_extend.cmo: (to be loaded either after "pa_o.cmo" or "pa_r.cmo") Syntax extensions for the Camlp4 grammar system (Chapter 4).
- pa_ocamllex.cma: (to be loaded after "pa_o.cmo" or "pa_r.cmo": ocamllex syntax (see section ??).
- pa_macro.cmo: add macros and IFDEF statements (conditional compilation) like C, in expressions or structure items. Add the options -D and -U to the camlp4 command line. To define the symbol FOO, either use "-DFOO" in the camlp4 command line, e.g.:

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

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

  ```
  Pa_macro.define "FOO";;
  ```

  compile it and load its corresponding object file:

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

  To undefine the symbol FOO, use either "-UFOO" or the function "Pa_macro.undef". The statement "IFDEF" can be used in a place of a structure item or an expression: in this last case, "ELSE" is mandatory:

  ```
  IFDEF FOO THEN type t = A of int END;;
  let f x = IFDEF FOO THEN x + 1 ELSE x - 1 END;;
  ```

- pa_ru.cmo: (to be loaded after "pa_r.cmo" only) Add type constraints of type unit inside sequences.

- Printing extensions:
  - pr_op.cmo: Try to rebuild parsers and streams in OCaml syntax.
  - pr_rp.cmo: Try to rebuild parsers and streams in revised syntax.
  - pr_extend.cmo: Try to rebuilt EXTEND instructions.
  - pr_depend.cmo: Print file dependencies.

Some shortcut commands are provided:

- "camlp4o" is equivalent to:

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

- "camlp4r" is equivalent to:

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

1.2 Preprocessing OCaml

1.2.1 The compiler

The standard use of Camlp4 is as a preprocessor for OCaml with the printer "`pr_dump.cmo`". The OCaml compiler "`ocamlc`" recognizes it as a dump, and does not apply another parsing. In case of typing errors, the error is normally showed in the input file.
We remind that the command "`ocaml`" has an option "`-pp`" for calling a preprocessor which automatically deals with intermediate files (see the documentation of OCaml). So, the normal use of the combination Camlp4-OCaml is done by the command:

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

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

1.2.2 Remark

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

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

However, there is a case where it is interesting to use "`pr_o.cmo`" for OCaml preprocessing. When the error is located in a quotation (chapter 3) or in a syntax extension (chapter 5), it is not always obvious to understand where the error is exactly located, and there may be no location for the error. Using temporarily camlp4 with "`pr_o.cmo`" allows to see the exact location of the error, which can be understood by the programmer of the quotation expander or the syntax extension.

1.2.3 The toplevel

The OCaml toplevel, "`ocaml`" can use Camlp4. In the toplevel, you can load:

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

To change the behavior of Camlp4, just load the needed module with the directives "`#load`" and "`#use`": for example, each load of "`pa_o.cmo`" or "`pa_r.cmo`" would change the current input syntax to be used in the next sentences.
The files included by the directive "`#use`" are treated by the current Camlp4 syntax too. Important remark: a syntax modification takes place after the complete load of a file, not after each sentence inside the file.
It is not possible to load a Camlp4 printing syntax file in the toplevel. The printing results are done by the OCaml toplevel in standard OCaml syntax.

1.3 User syntax modifications

The next chapters explain the different ways to create files holding syntax modifications. The simplest one is the quotations system (chapter 3).

To introduce other syntax extensions of the language, or to redefine the complete language syntax, one must use the Camlp4 Grammar System (chapter 4). The following chapter shows how to extend or redefine the language syntax (chapter 5).

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

Chapter 2

Streams and parsers

Objective Caml comprises a library type for streams (possibly infinite sequences of elements, that are evaluated on demand), and associated stream expressions, to build streams, and stream patterns, to destructure streams. Streams and stream patterns provide a natural approach to the writing of recursive-descent parsers.

Streams are presented by the following extensions to the syntactic classes of expressions:

```
            expr ::= ...
                   | [< >]
                   | [< stream-component  { ; stream-component } >]
                   | parser [pattern]  stream-matching
                   | match expr with parser  [pattern]  stream-matching
stream-component ::=
                     ' expr
                   | expr
 stream-matching ::= stream-pattern [pattern] -> expr
                         { | stream-pattern [pattern] ->  expr }
  stream-pattern ::=
                     [< >]
                   | [< stream-pat-comp
                           { ; stream-pat-comp  [ ?? expr ] }>]
 stream-pat-comp ::=
                     ' pattern  [ when expr ]
                   | pattern =  expr
                   | ident
```

Stream expressions are bracketed by `[<` and `>]`. They represent the concatenation of their components. The component ' expr represents the one-element stream whose element is the value of expr. The component expr represents a sub-stream. For instance, if both s and t are streams of integers, then `[<'1; s; t; '2>]` is a stream of integers containing the element 1, then the elements of s, then those of t, and finally 2. The empty stream is denoted by `[< >]`.

Unlike any other kind of expressions in the language, stream expressions are submitted to lazy evaluation: the components are not evaluated when the stream is built, but only when they are accessed during stream matching. The components are evaluated once, the first time they are accessed; the following accesses reuse the value computed the first time.

Stream patterns, also bracketed by `[<` and `>]`, describe initial segments of streams. In particular, the stream pattern `[< >]` matches all streams. Stream pattern components are matched against the corresponding elements of a stream. The component `' pattern` matches the corresponding stream element against the pattern; if followed by when, the match is accepted only if the result of the guard expression is true. The component `pattern = expr` applies the function denoted by expr to the current stream, then matches the result of the function against pattern. Finally, the component ident simply binds the identifier to the stream being matched.

Stream matching proceeds destructively: once a component has been matched, it is discarded from the stream (by in-place modification).

Stream matching proceeds in two steps: first, a pattern is selected by matching the stream against the first components of the stream patterns; then, the following components of the selected pattern are checked against the stream. If the following components do not match, the exception Stream.Error is raised. There is no backtracking here: stream matching commits to the pattern selected according to the first element. If none of the first components of the stream patterns match, the exception Stream.Failure is raised. The Stream.Failure exception causes the next alternative to be tried, if it occurs during the matching of the first element of a stream, before matching has committed to one pattern.

The streams hold the count of their elements discarded. The optional pattern before the first stream pattern is bound to the stream count before the matching. The one after each stream pattern (optional, too) is bound to the stream count after the matching.

The exception Stream.Error has a string parameter coming from the optional `?? expr` after the stream pattern components (its default is the empty string). This expression is evaluated only in case of error.

Chapter 3

Quotations

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

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

There are two kinds of quotation expanders:

- Those returning *strings*. The result of the expander is parsed by Camlp4 afterwards. They are easy to use, but they depend on the language syntax: the returned string must be of `OCaml` syntax if the language syntax used is the normal syntax, in revised syntax if the language syntax is the revised syntax, etc.
- Those returning *syntax trees*. They are independent from the language syntax used, but to use them, one muse know how to create syntax tree nodes. See appendix A.

3.1 Syntax

There are two kinds of quotations:

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

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

3.2 Creating a quotation expander

A quotation expander is a function written in OCaml. A call to the Camlp4 library function "`Quotation.add`" (see section **??**) adds the quotation expander. The variable

"`Quotation.default`" holds the name of the default quotation.

The file holding the quotation expander must be compiled using the OCaml compiler, with the *Camlp4 library directory* in its directory path (option "`-I`"). An object file (ending with "`.cmo`") is created, which is loadable in Camlp4, as parameter of the command "camlp4", or in the OCaml toplevel using the directive "`#load`".

3.3 Antiquotations

Antiquotation is a programming technique to specify expressions or patterns in a quotation, parsed in the context of the enclosing expression or pattern where the quotation is.

The way an antiquotation is specified depends on the quotation expander: typically a specific character and an identifier, or specific parentheses and a complete expression or pattern.

3.4 Errors

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

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

3.4.1 Exception raised in an expander

If an exception is raised in the expander, the quotation is highlighted, the exception name is displayed, and the parsing fails.

To be more precise, the expander can call the function "`Stdpp.raise_with_loc`" instead instead of "`raise`", this first function having a supplementary parameter which is the location of the error relative to the begin of the quotation. In this case, the error message highlights only the located part of the quotation.

3.4.2 Error while parsing the expander result (string expanders)

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

To solve this problem, a special option of Camlp4 can be used: "`-QD`" (see the manual page). It is also possible to set the variable "`Pcaml.quotation_dump_file`" (see section **??**). The offending string is then written in a file (whose name is specified by this option or this variable) and the error message gives the location of the error in this file. This technique is useful when perfecting or debugging the quotation expander: a good quotation expander should never generate errored strings.

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

3.4.3 Typing error

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

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

  $$\textit{location\_directive} \ \texttt{::=} \ \texttt{\$} \ \textit{int} \ \texttt{:} \ \textit{string} \ \texttt{\$}$$

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

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

3.5 Predefined quotation expanders

A quotation expander is provided in the file "`q_MLast.cmo`" allowing to create abstract syntax tree nodes from concrete revised syntax. It is used in language syntax extensions (chapter 5).

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

Chapter 4

Grammars

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

4.1 Grammars and entries

The Camlp4 library provides a module "`Grammar`" and a syntax extension file "`pa_extend.cmo`" to program grammars. All details on modules, types and functions are described chapter 8.
A grammar can be created using the function "`Grammar.gcreate`". It takes a lexer as parameter. A good candidate it the function "`Plexer.gmake`" but the user can creates its own lexer, providing that it is of type "`Token.glexer`".
A grammar holds entries. An entry can be created using the function "`Grammar.Entry.create`". The first parameter is the grammar, the second one is a string used to name the entry in error messages and in entries dump. Entries are created empty, i.e. raising "`Stream.Error`" when called. An entry is composed of entry precedence levels, the first one being the least precedent and the last one the most.
Lastly, a stream can be parsed by en entry using the function "`Grammar.Entry.parse`". In case of syntax error, the exception "`Stream.Error`" is raised, encapsulated by the exception "`Stdpp.Exc_located`", giving the location of the error.

4.2 Extension

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

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

`EXTEND`, `GLOBAL` and `END` are keywords. There are some other keywords in this instruction, all in uppercase.
The *entry* is a name of an entry variable (simple identifier or prefixed by a module name). It is extended according to the rules given by *extension*. The optional *position* tells how to insert the extension among the entry precedence levels: by default, the extension is done at the first precedence level (creating this level if the entry is empty).
The optional *global-list* restricts the list of the *global* entries (separated by spaces) extended in the "`EXTEND`" instruction. The other entries are created locally. By default, all entries are global and must correspond to entry variables visible at the "`EXTEND`" instruction point.

Syntax of a position

A *position* can be:

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

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

Syntax of an extension

The syntax of an entry *extension* is:

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

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

Syntax of a level-rules

The *level-rules* have the format:

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

The *pattern* is a language simple pattern, limited to identifiers, wildcard (character "underscore") and tuples of patterns. If not present, it is equivalent to the wildcard.
The *action* is a language expression where the patterns are bound to the result of their corresponding *symbol*; in addition, the variable "`loc`" is bound to the source location of the rule. The action part is optional; by default it is the value "`()`".

A *symbol* is either:

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

An entry name (simple identifier or prefixed by a module name) meaning a call to this entry. Its type is the entry result type. The entry name can be followed by the keyword `LEVEL` and a string, meaning a direct call to that level.

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

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

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

A metasymbol among:

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

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

  meaning a list (potentially empty for `LIST0` and with at least one element for `LIST1`) of *symbol1*, whose elements are optionally separated by *symbol2*. The type is `t1 list` where `t1` is the type of *symbol1* (the result of the optional *symbol2* is lost).
- `OPT` followed by a *symbol*, meaning this symbol or nothing. The type is `t option` where `t` is the type of *symbol*.

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

A symbol between parentheses

4.3 Parsed language

Entries are stream parsers improved. Streams use recursive descendant parsing (something close to LL(1)). Each precedence level can be seen as an independent extensible stream parser.

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

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

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

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

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

4.4 Deletion

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

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

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

4.5 Writing a lexer

The lexers have to create tokens of any type you want. In the lexer interface (see below), you have to provide the way a token pattern (which is of type `(string * string)`) has to be compared with a token value of your token type.

A simple token type is `(string * string)` like the token pattern type. Its advantage is that a default function is provided in the module `Token` for the way the token patterns are parsed.

The token patterns are defined in the `EXTEND` statement, as terminal symbols in the rules. In a rule, an identifier in uppercase character *not* among the reserved ones (LIST0, OPT, and so on) is a token pattern. The identifier is the first string in the token pattern type. If the symbol is followed by a string, it is the second string of the token pattern type; if not, the string is empty. Moreover a string alone is also a token pattern with the first string being empty.

Examples: the following symbol rules in an EXTEND statement:

```
IDENT "bar"
INT "32"
x = IDENT
i = INT
"let"
```

correspond (resp.) to the following token patterns:

```
("IDENT", "bar")
("INT", "32")
("IDENT", "")
("INT", "")
("", "let")
```

A lexer appropriate for the `Grammar.gcreate` function is a record of type `Token.glexer` which is a record with the following fields:

- `tok_func`: it is the main lexing function. It is called once when calling `Grammar.Entry.parse` with the input character stream. The simplest way to create this function is to use either the function `Token.lexer_func_of_parser` if you use a stream parser, or `Token.lexer_func_of_ocamllex` if you prefer use an ocamllex function. Notice that in both cases, your lexer function must return a couple of a token value (of your token type) and a token location (a couple of integers).
- `tok_using`: it is a function taking a token pattern as parameter. When `EXTEND` statement first scans all symbols in all rules, it calls this function with the token patterns encountered (a token pattern is of type `(string * string)`). This function allows you to check the pattern (that the constructor is among the ones the lexer generates: raise an exception if not) and enter keywords (if you have keywords) in your keyword table (if you use a keyword table).
- `tok_removing`: like `tok_using` but used when a rule is removed.
- `tok_match`: tells how a token pattern is parsed. It must return a function taking a token as parameter, returning either the resulting string or the exception `Stream.Failure`. Use the function `Token.default_match` for a default behavior when your token type is `(string * string)`. Remark: for efficiency, if you write your own version, write is as a function taking a token pattern and returning a function for each case, *not* as a function with two parameters.
- `tok_text`: give the name of a token pattern to be used in error messages. You can use the function named `lexer_text` provided in the module `Token`.

Chapter 5

Syntax extensions

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

5.1 The abstract syntax tree

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

5.2 Extending the language

The language predefined syntaxes, “`pa_o.cmo`” and “`pa_r.cmo`” use the Camlp4 grammar system: so they are extensible.
The module “`Pcaml`” holds grammars and entries variables used by the language predefined syntaxes files. Syntax extensions can be done by extensions of the appropriate entries.
In some entries, “`pa_o.cmo`” and “`pa_r.cmo`” define labels allowing to locate some levels and to add new rules relative to these levels. The entries and their labels are:

- “`Pcaml.expr`”, for expressions, with the labels:
  - “`let`” corresponding to the level of the instruction “`let...in`”.
  - “`||`”, “`&&`”, “`<`”, “`+`”, “`*`”, “`**`” corresponding to the so named operator.
  - “`apply`” corresponding to the curryfied application.
  - “`simple`” corresponding to simple expressions.
- “`Pcaml.patt`”, for patterns, with the label “`simple`” corresponding to simple patterns.
- “`Pcaml.ctyp`”, for types, with the label “`simple`” corresponding to simple types.
- “`Pcaml.sig_item`” and “`Pcaml.str_item`”, for signature and structure items, with only one level labelled “`top`”.
- “`Pcaml.module_type`” and “`Pcaml.module_expr`”, for module types and expressions.
- Some other labels exist but are not documented.

5.3 Examples of language extensions

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

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

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

5.3.1 Infix

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

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

5.3.2 Repeat until à la Pascal

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

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

5.4 Redefining the whole syntax

To redefine the whole syntax, one must start with an empty grammar and empty entries and extend them. The library module "`Grammar.Unsafe`" hold functions to clean up the grammars and the entries.
Examples can be found in the distribution: the files "`meta/pa_r.ml`" and "`etc/pa_o.ml`" which are the sources of "`pa_r.cmo`" and "`pa_o.cmo`".

Chapter 6

The Revised syntax

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

6.1 Phrases

- A simple semicolon ends the sentences, in structures and in signatures. The double semicolon is not a token. There is no ambiguity with the sequence, which has a special construction (see 6.2).
- The declaration of a global variable is introduced by the keyword "`value`", "`let`" being reserved to the construction "`let..in`":

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

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

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

6.2 Imperative constructions

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

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

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

OCaml	Revised

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

6.3 Tuples and lists

- Parentheses are mandatory in tuples:

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

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

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

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

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

Note the two ways to write the last case.

6.4 Irrefutable patterns

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

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

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

6.5 Constructions with matching

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

OCaml	Revised

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

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

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

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

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

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

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

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

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

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

must be written in Revised:

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

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

6.6 Mutables and assignment

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

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

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

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

6.7 Types

- 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 mandatory 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:

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

- The predefined constructors "`True`" and "`False`" start with an uppercase letter.
- In record types, the keyword "`mutable`" must appear after the colon:

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

6.8 Modules

Modules application uses curryfication:

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

6.9 Objects

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

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

6.10 Miscellaneous

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

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

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

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

- No more "`begin end`" construction. One must use parentheses.

- The operators as functions are written with an backslash:

OCaml	Revised
`(+)` `(mod)`	`\+` `\mod`

- The operators with special characters are not automatically infix. To define infixes, use the syntax extensions (chapter 5).

- It is possible to group together several declarations either in an interface or in an implementation by enclosing them between "`declare`" and "`end`". Example in an interface:

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

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

6.11 Streams and parsers

- The streams and the stream patterns are bracketed with "`[:`" and "`:]`" instead of "`[<`" and "`>]`".

- The stream component "terminal" is written with a backquote instead of a quote:

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

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

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

Chapter 7

Other extensions

7.1 Extensible functions

Camlp4 provides a system of functions defined by pattern matching, which are extensible.

The library module is `Extfun` and the syntax to be loaded is `"pa_extfun.cmo"`. The empty function is `Extfun.empty`. You can extend a function using the statement `extfun` whose syntax is:

`extfun` *expression* `with`
`[` *pattern-1* `->` *expression-1*
`|` *pattern-2* `->` *expression-2*
...
`|` *pattern-n* `->` *expression-n* `]`

The patterns are ordered in lexicographic order (for example, in a tuple, first comparing the first elements of the tuple). Variables are inserted after constructors. In an extension, the patterns do not need to be in a "good" order, since they are sorted. Non exhaustive pattern matching do not generate a warning.

"Or" patterns can be used only at the first level. In this case:

```
pat1 | pat2 -> expr
```

The binding is split into two cases (the expr is duplicated):

```
  pat1 -> expr
| pat2 -> expr
```

Internal "or" patterns inside patterns are not accepted.

The statement `extfun` returns another extensible function. The type of extensible functions is `('a, 'b) Extfun.t`. To use an extensible function, one must use the function `Extfun.apply` which transforms it in a function of type `'a -> 'b`. If matching failed, such a function raises the exception `Extfun.Failure`.

The contents (patterns) of an extensible function can be displayed using `Extfun.print`.

Remark: extensible functions are not efficient: when applied, all patterns are tested, one by one, until one of them matches.

Extensible functions are used in Camlp4 extensible pretty printing.

7.2 Functional streams

The functional streams are another implementation of streams. Like normal streams, their contents can be accessible only one element at a time. But the elements are not removed. A functional stream parser returns the couple of a result and the remaining stream.

The library module is `Fstream` and the syntax to be loaded is `"pa_fstream.cmo"`. The syntax of a functional stream is:

functional-stream ::=
  `fstream [:` *list-of-components-separated-by-semicolon* `:]`
component ::=
    ‘ *stream-element*
  | *stream*

and a functional parser, applying to a functional stream is:

functional-parser ::=
  `fparser`
  `[` *stream-pattern-1* `->` *expression-1*
  `|` *stream-pattern-2* `->` *expression-2*
  ⋮
  `|` *stream-pattern-n* `->` *expression-n* `]`
stream-pattern ::=
  `[:` *list-of-components-separated-by-semicolon* `:]`
component ::=
    ‘ *stream-pattern-element*
  | *pattern* `=` *expression*
  | *stream-pattern*

The functional stream patterns elements syntax are actually the same than in normal stream pattern.

A functional stream is of type `'a Fstream.t` and a functional stream parser of type `'a Fstream.t -> ('a * 'a Fstream.t) option`. When a parser fails, it returns `None`, otherwise `Some` of the result and the remaining stream. The elements in the initial stream are not removed.

A functional parser use limited backtrack. It is a backtrack in a sense that when a rule fails, the next rule is tested with the initial stream. If no rule applies, the functional parser returns `None`. There is no Error exception causing the parsing to be abandoned.

The backtrack is limited in a sense that if a rule is `[: p1 = e1; p2 = e2 :]`, if `e2` fails, the rule is abandoned: there is no attempt to try the next possible rule inside `e1` (which would suppose continuations).

The functions available in the module `Fstream` are like the ones in `Stream`. But there is no function "Fstream.peek", only `Fstream.next`.

Functional parsers have a drawback that in case of syntax error, one cannot know where, since the parsing continues until all rules have been tested. To turn around this problem, the function `Fstream.count_frozen` returns the number of unfrozen tokens in the stream, allowing to find the location of the error, providing a location array have been used (which is normal usage in stream parsing and grammars). It works if the stream had not been unfrozen before.

Chapter 8

Camlp4 library modules

When linking an application using this library modules, you have to add the file `gramlib.cma` in the command line. For example, with the line:

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

8.1 Module `MLast`: abstract syntax tree

This library module is accessible, but not documented. The creation of abstract syntax tree nodes must be done using quotations, defined in the file `q_MLast.cmo`. See appendix A.
However, some functions are provided to access the "location" field in nodes. They are:

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

8.2 Module

```
(** Extensible grammars.

    This module implements the Camlp4 extensible grammars system.
    Grammars entries can be extended using the [EXTEND] statement,
    added by loading the Camlp4 [pa_extend.cmo] file. *)
```

```
type g
   (** The type for grammars, holding entries. *)
val gcreate : Token.t Token.glexer -> g
   (** Create a new grammar, without keywords, using the lexer given
       as parameter. *)
val tokens : g -> string -> (string * int) list
   (** Given a grammar and a token pattern constructor, returns the list of
       the corresponding values currently used in all entries of this grammar.
       The integer is the number of times this pattern value is used.

       Examples:
-      If the associated lexer uses ("", xxx) to represent a keyword
       (what is represented by then simple string xxx in an [EXTEND]
       statement rule), the call [Grammar.token g ""] returns the keywords
       list.
-      The call [Grammar.token g "IDENT"] returns the list of all usages
       of the pattern "IDENT" in the [EXTEND] statements. *)
val glexer : g -> Token.t Token.glexer
   (** Return the lexer used by the grammar *)

module Entry :
  sig
    type 'a e
    val create : g -> string -> 'a e
    val parse : 'a e -> char Stream.t -> 'a
    val parse_token : 'a e -> Token.t Stream.t -> 'a
    val name : 'a e -> string
    val of_parser : g -> string -> (Token.t Stream.t -> 'a) -> 'a e
    val print : 'a e -> unit
    val find : 'a e -> string -> Obj.t e
    external obj : 'a e -> Token.t Gramext.g_entry = "%identity"
  end
   (** Module to handle entries.
-       [Entry.e] is the type for entries returning values of type ['a].
-       [Entry.create g n] creates a new entry named [n] in the grammar [g].
-       [Entry.parse e] returns the stream parser of the entry [e].
-       [Entry.parse_token e] returns the token parser of the entry [e].
-       [Entry.name e] returns the name of the entry [e].
-       [Entry.of_parser g n p] makes an entry from a token stream parser.
-       [Entry.print e] displays the entry [e] using [Format].
-       [Entry.find e s] finds the entry named [s] in [e]'s rules.
-       [Entry.obj e] converts an entry into a [Gramext.g_entry] allowing
-      to see what it holds ([Gramext] is visible, but not documented). *)

val of_entry : 'a Entry.e -> g
   (** Return the grammar associated with an entry. *)
```

```
(** {6 Clearing grammars and entries} *)

module Unsafe :
  sig
    val gram_reinit : g -> Token.t Token.glexer -> unit
    val clear_entry : 'a Entry.e -> unit
  end
   (** Module for clearing grammars and entries. To be manipulated with
       care, because: 1) reinitializing a grammar destroys all tokens
       and there may have problems with the associated lexer if it has
       a notion of keywords; 2) clearing an entry does not destroy the
       tokens used only by itself.
-      [Unsafe.reinit_gram g lex] removes the tokens of the grammar
-      and sets [lex] as a new lexer for [g]. Warning: the lexer
-      itself is not reinitialized.
-      [Unsafe.clear_entry e] removes all rules of the entry [e]. *)

(** {6 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 val lexer : te Token.glexer end
   (** The input signature for the functor [Grammar.GMake]: [te] is the
       type of the tokens. *)

module type S =
  sig
    type te
    type parsable
    val parsable : char Stream.t -> parsable
    val tokens : string -> (string * int) list
    val glexer : te Token.glexer
    module Entry :
      sig
        type 'a e
        val create : string -> 'a e
        val parse : 'a e -> parsable -> 'a
        val parse_token : 'a e -> te Stream.t -> 'a
        val name : 'a e -> string
        val of_parser : string -> (te Stream.t -> 'a) -> 'a e
        val print : 'a e -> unit
        external obj : 'a e -> te Gramext.g_entry = "%identity"
      end
    module Unsafe :
```

```
      sig
        val gram_reinit : te Token.glexer -> unit
        val clear_entry : 'a Entry.e -> unit
      end
    val extend :
      'a Entry.e -> Gramext.position option ->
        (string option * Gramext.g_assoc option *
           (te Gramext.g_symbol list * Gramext.g_action) list)
          list ->
        unit
    val delete_rule : 'a Entry.e -> te Gramext.g_symbol list -> unit
  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

(** {6 Miscellaneous} *)

val error_verbose : bool ref
   (** Flag for displaying more information in case of parsing error;
       default = [False] *)

val warning_verbose : bool ref
   (** Flag for displaying warnings while extension; default = [True] *)

val strict_parsing : bool ref
   (** Flag to apply strict parsing, without trying to recover errors;
       default = [False] *)

val print_entry : Format.formatter -> 'te Gramext.g_entry -> unit
   (** General printer for all kinds of entries (obj entries) *)

val iter_entry : ('te Gramext.g_entry -> unit) -> 'te Gramext.g_entry -> 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. *)

val fold_entry :
```

```
  ('te Gramext.g_entry -> 'a -> 'a) -> 'te Gramext.g_entry -> '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. *)
```

8.3 Module

```
(** Extensible functions.

   This module implements pattern matching extensible functions.
   To extend, use syntax [pa_extfun.cmo]:

      [extfun e with [ pattern_matching ]] *)

type ('a, 'b) t
   (** The type of the extensible functions of type ['a -> 'b] *)
val empty : ('a, 'b) t
   (** Empty extensible function *)
val apply : ('a, 'b) t -> 'a -> 'b
   (** Apply an extensible function *)
exception Failure
   (** Match failure while applying an extensible function *)
val print : ('a, 'b) t -> unit
   (** Print patterns in the order they are recorded *)
```

8.4 Module

```
(* Module [Fstream]: functional streams *)

(* 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 have 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 'a t
    (* The type of 'a functional streams *)
val from : (int -> 'a option) -> 'a t
    (* [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. *)

val of_list : 'a list -> 'a t
    (* Return the stream holding the elements of the list in the same
       order. *)
val of_string : string -> char t
    (* Return the stream of the characters of the string parameter. *)
val of_channel : in_channel -> char t
    (* Return the stream of the characters read from the input channel. *)

val iter : ('a -> unit) -> 'a t -> unit
    (* [Fstream.iter f s] scans the whole stream s, applying function [f]
       in turn to each stream element encountered. *)

val next : 'a t -> ('a * 'a t) option
    (* 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. *)
val empty : 'a t -> (unit * 'a t) option
    (* Return [Some ((), s)] if the stream is empty where [s] is itself,
       else [None] *)
val count : 'a t -> int
    (* Return the current count of the stream elements, i.e. the number
       of the stream elements discarded. *)
val count_unfrozen : 'a t -> int
    (* Return the number of unfrozen elements in the beginning of the
       stream; useful to determine the position of a parsing error (longuest
       path). *)
```

8.5 Module

```
(** Language grammar, entries and printers.

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

val syntax_name : string ref
```

```
(** {6 Parsers} *)

val parse_interf :
  (char Stream.t -> (MLast.sig_item * MLast.loc) list * bool) ref
val parse_implem :
  (char Stream.t -> (MLast.str_item * MLast.loc) list * bool) ref
   (** 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 that 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 Camlp4 extended grammars. By
       default, they use the grammars entries [implem] and [interf]
       defined below. *)

val gram : Grammar.g
   (** Grammar variable of the OCaml language *)

val interf : ((MLast.sig_item * MLast.loc) list * bool) Grammar.Entry.e
val implem : ((MLast.str_item * MLast.loc) list * bool) Grammar.Entry.e
val top_phrase : MLast.str_item option Grammar.Entry.e
val use_file : (MLast.str_item list * bool) Grammar.Entry.e
val module_type : MLast.module_type Grammar.Entry.e
val module_expr : MLast.module_expr Grammar.Entry.e
val sig_item : MLast.sig_item Grammar.Entry.e
val str_item : MLast.str_item Grammar.Entry.e
val expr : MLast.expr Grammar.Entry.e
val patt : MLast.patt Grammar.Entry.e
val ctyp : MLast.ctyp Grammar.Entry.e
val let_binding : (MLast.patt * MLast.expr) Grammar.Entry.e
val type_declaration : MLast.type_decl Grammar.Entry.e
val class_sig_item : MLast.class_sig_item Grammar.Entry.e
val class_str_item : MLast.class_str_item Grammar.Entry.e
val class_expr : MLast.class_expr Grammar.Entry.e
val class_type : MLast.class_type Grammar.Entry.e
   (** Some entries of the language, set by [pa_o.cmo] and [pa_r.cmo]. *)

val input_file : string ref
   (** The file currently being parsed. *)
val output_file : string option ref
   (** The output file, stdout if None (default) *)
val report_error : exn -> unit
   (** Prints an error message, using the module [Format]. *)
val quotation_dump_file : string option ref
```

```
   (** [quotation_dump_file] optionally tells the compiler to dump the
       result of an expander if this result is syntactically incorrect.
       If [None] (default), this result is not dumped. If [Some fname], the
       result is dumped in the file [fname]. *)
val version : string
   (** The current version of Camlp4. *)
val add_option : string -> Arg.spec -> string -> unit
   (** Add an option to the command line options. *)
val no_constructors_arity : bool ref
   (** [True]: dont generate constructor arity. *)
(*value no_assert : ref bool;
   (** [True]: dont generate assertion checks. *)
*)

val sync : (char Stream.t -> unit) ref

val handle_expr_quotation : MLast.loc -> string * string -> MLast.expr
val handle_expr_locate : MLast.loc -> int * string -> MLast.expr

val handle_patt_quotation : MLast.loc -> string * string -> MLast.patt
val handle_patt_locate : MLast.loc -> int * string -> MLast.patt

val expr_reloc : (MLast.loc -> MLast.loc) -> int -> MLast.expr -> MLast.expr
val patt_reloc : (MLast.loc -> MLast.loc) -> int -> MLast.patt -> MLast.patt

(** To possibly rename identifiers; parsers may call this function
    when generating their identifiers; default = identity *)
val rename_id : (string -> string) ref

(** Allow user to catch exceptions in quotations *)
type err_ctx =
    Finding
  | Expanding
  | ParsingResult of (int * int) * string
  | Locating
exception Qerror of string * err_ctx * exn

(** {6 Printers} *)

open Spretty

val print_interf : ((MLast.sig_item * MLast.loc) list -> unit) ref
val print_implem : ((MLast.str_item * MLast.loc) list -> unit) ref
   (** Some printers, set by [pr_dump.cmo], [pr_o.cmo] and [pr_r.cmo]. *)

type 'a printer_t =
  { mutable pr_fun : string -> 'a -> string -> kont -> pretty;
```

```
    mutable pr_levels : 'a pr_level list }
and 'a pr_level =
  { pr_label : string;
    pr_box : 'a -> pretty Stream.t -> pretty;
    mutable pr_rules : 'a pr_rule }
and 'a pr_rule =
  ('a, ('a curr -> 'a next -> string -> kont -> pretty Stream.t)) Extfun.t
and 'a curr = 'a -> string -> kont -> pretty Stream.t
and 'a next = 'a -> string -> kont -> pretty
and kont = pretty Stream.t

val pr_sig_item : MLast.sig_item printer_t
val pr_str_item : MLast.str_item printer_t
val pr_module_type : MLast.module_type printer_t
val pr_module_expr : MLast.module_expr printer_t
val pr_expr : MLast.expr printer_t
val pr_patt : MLast.patt printer_t
val pr_ctyp : MLast.ctyp printer_t
val pr_class_sig_item : MLast.class_sig_item printer_t
val pr_class_str_item : MLast.class_str_item printer_t
val pr_class_type : MLast.class_type printer_t
val pr_class_expr : MLast.class_expr printer_t

val pr_expr_fun_args :
  (MLast.expr, (MLast.patt list * MLast.expr)) Extfun.t ref

val find_pr_level : string -> 'a pr_level list -> 'a pr_level

val top_printer : 'a printer_t -> 'a -> unit
val string_of : 'a printer_t -> 'a -> string

val inter_phrases : string option ref
```

8.6 Module

```
(** A lexical analyzer. *)

val gmake : unit -> Token.t Token.glexer
   (** Some lexer provided. See the module [Token]. The tokens returned
       follow the Objective Caml and the Revised syntax lexing rules.

       The meaning of the tokens are:
-      * [("", s)] is the keyword [s].
-      * [("LIDENT", s)] is the ident [s] starting with a lowercase letter.
-      * [("UIDENT", s)] is the ident [s] starting with an uppercase letter.
```

```
-      * [("INT", s)] (resp. ["INT32"], ["INT64"] and ["NATIVEINT"])
         is an integer constant whose string source is [s].
-      * [("FLOAT", s)] is a float constant whose string source is [s].
-      * [("STRING", s)] is the string constant [s].
-      * [("CHAR", s)] is the character constant [s].
-      * [("QUOTATION", "t:s")] is a quotation [t] holding the string [s].
-      * [("ANTIQUOT", "t:s")] is an antiquotation [t] holding the string [s].
-      * [("LOCATE", "i:s")] is a location directive at pos [i] holding [s].
-      * [("EOI", "")] is the end of input.

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

       Warning: the string associated with the constructor [STRING] 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 character "return", but a string of two
       elements: the backslash and the character ["n"]. To interpret
       a string use the function [Token.eval_string]. Same thing for
       the constructor [CHAR]: to get the character, don't get the
       first character of the string, but use the function
       [Token.eval_char].

       The lexer do not use global (mutable) variables: instantiations
       of [Plexer.gmake ()] do not perturb each other.  *)

val dollar_for_antiquotation : bool ref
   (** When True (default), the next call to [Plexer.make ()] returns a
       lexer where the dollar sign is used for antiquotations. If False,
       the dollar sign can be used as token. *)

val specific_space_dot : bool ref
   (** When False (default), the next call to [Plexer.make ()] returns a
       lexer where the dots can be preceded by spaces. If True, dots
       preceded by spaces return the keyword " ." (space dot), otherwise
       return the keyword "." (dot). *)

val no_quotations : bool ref
   (** When True, all lexers built by [Plexer.make ()] do not lex the
       quotation syntax any more. Default is False (quotations are
       lexed). *)
```

8.7 Module

```
(** Quotation operations. *)

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

(** The type for quotation expanders kind:
-       [ExStr exp] for an expander [exp] returning a string which
          can be parsed to create a syntax tree. Its boolean parameter
          tells whether the quotation is in position of an expression
          (True) or in position of a pattern (False). Quotations expanders
          created with this way may work for some particular language syntax,
          and not for another one (e.g. may work when used with Revised
          syntax and not when used with Ocaml syntax, and conversely).
-       [ExAst (expr_exp, patt_exp)] for expanders returning directly
          syntax trees, therefore not necessiting to be parsed afterwards.
          The function [expr_exp] is called when the quotation is in
          position of an expression, and [patt_exp] when the quotation is
          in position of a pattern. Quotation expanders created with this
          way are independant from the language syntax. *)

val add : string -> expander -> unit
   (** [add name exp] adds the quotation [name] associated with the
       expander [exp]. *)

val find : string -> expander
   (** [find name] returns the expander of the given quotation name. *)

val default : string ref
   (** [default] holds the default quotation name. *)

val translate : (string -> string) ref
   (** function translating quotation names; default = identity *)
```

8.8 Module

```
(** Standard definitions. *)

exception Exc_located of (int * int) * exn
   (** [Exc_located loc e] is an encapsulation of the exception [e] with
       the input location [loc]. To be used in quotation expanders
       and in grammars to specify some input location for an error.
       Do not raise this exception directly: rather use the following
```

```
       function [raise_with_loc]. *)

val raise_with_loc : int * int -> exn -> 'a
   (** [raise_with_loc loc e], if [e] is already the exception [Exc_located],
       re-raise it, else raise the exception [Exc_located loc e]. *)

val line_of_loc : string -> int * int -> 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. *)

val loc_name : string ref
   (** Name of the location variable used in grammars and in the predefined
       quotations for OCaml syntax trees. Default: [loc] *)
```

8.9 Module

```
(** Lexers for Camlp4 grammars.

   This module defines the Camlp4 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 [Glexer] one day). *)

type pattern = string * string
    (** Token patterns come from the EXTEND statement.
-       The first string is the constructor name (must start with
        an uppercase character). When it is empty, the second string
        is supposed to be a keyword.
-       The second string is the constructor parameter. Empty if it
        has no parameter.
-       The way tokens patterns are interpreted to parse tokens is
        done by the lexer, function [tok_match] below. *)

exception Error of string
    (** An lexing error exception to be used by lexers. *)

(** {6 Lexer type} *)

type location = int * int
type location_function = int -> location
  (** The type for a function associating a number of a token in a stream
      (starting from 0) to its source location. *)
type 'te lexer_func = char Stream.t -> 'te Stream.t * location_function
  (** The type for a lexer function. The character stream is the input
```

```
      stream to be lexed. The result is a pair of a token stream and
      a location function for this tokens stream. *)

type 'te glexer =
  { tok_func : 'te lexer_func;
    tok_using : pattern -> unit;
    tok_removing : pattern -> unit;
    tok_match : pattern -> 'te -> string;
    tok_text : pattern -> string;
    mutable tok_comm : location list option }
   (** The type for a lexer used by Camlp4 grammars.
-       The field [tok_func] is the main lexer function. See [lexer_func]
       type above. This function may be created from a [char stream parser]
       or for an [ocamllex] function using the functions below.
-       The field [tok_using] is a function telling the lexer that the grammar
       uses this token (pattern). The lexer can check that its constructor
       is correct, and interpret some kind of tokens as keywords (to record
       them in its tables). Called by [EXTEND] statements.
-       The field [tok_removing] is a function telling the lexer that the
       grammar does not uses the given token (pattern) any more. If the
       lexer has a notion of "keywords", it can release it from its tables.
       Called by [DELETE_RULE] statements.
-       The field [tok_match] is a function taking a pattern and returning
       a function matching a token against the pattern. Warning: for
       efficency, write it as a function returning functions according
       to the values of the pattern, not a function with two parameters.
-       The field [tok_text] returns the name of some token pattern,
       used in error messages.
-       The field [tok_comm] if not None asks the lexer to record the
       locations of the comments.  *)

val lexer_text : pattern -> string
   (** A simple [tok_text] function for lexers *)

val default_match : pattern -> string * string -> string
   (** A simple [tok_match] function for lexers, appling to token type
       [(string * string)] *)

(** {6 Lexers from char stream parsers or ocamllex function}

   The functions below create lexer functions either from a [char stream]
   parser or for an [ocamllex] function. With the returned function [f],
   the simplest [Token.lexer] can be written:
   {[
          { 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. *)

val lexer_func_of_parser : (char Stream.t -> 'te * location) -> 'te lexer_func
   (** A lexer function from a lexer written as a char stream parser
       returning the next token and its location. *)
val lexer_func_of_ocamllex : (Lexing.lexbuf -> 'te) -> 'te lexer_func
   (** A lexer function from a lexer created by [ocamllex] *)

val make_stream_and_location :
  (unit -> 'te * location) -> 'te Stream.t * location_function
   (** General function *)

(** {6 Useful functions} *)

val eval_char : string -> char
val eval_string : 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)]
       returns [c] and [Token.eval_string (String.escaped s)] returns [s] *)
```

Appendix A

Quotations for creating abstract syntax trees

The file "`q_MLast.cmo`" in Camlp4 library directory provides some quotations expanders to create abstract syntax tree nodes using a concrete syntax.
The types generated are defined in the module "`MLast`" provided in the library, but not documented, for the normal usage is to use these quotations.

A.1 General rules

The syntax used in quotations use the revised concrete syntax (chapter 6) with a system of "antiquotations" to insert expressions or patterns of the language in the concrete syntax, to make all possible values of the syntax tree nodes. These antiquotations have the format:

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

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

All quotations generate values using the variable named "`loc`": if this variable is not defined, the `OCaml` compiler complains. It is the source location of the node (a couple of integers: first char - last char plus one). If there is no associated source of the node, just define it as `(0, 0)`. In semantic actions of Camlp4 extensible grammars, the variable `loc` is defined (by the system).

This chapter defines only the basic syntax trees with the basic antiquotations. But you can write any complicated program in them, providing it is of the good type.

A.2 Quotation `expr`

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

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

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

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

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

<:expr< [| $list:el$ |] >>: array; "el" is the list of expressions of the array.

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

<:expr< $chr:c$ >>: character constant; "c" is a string representing the character, like would be the result of the OCaml function Char.escaped.

<:expr< (e :> t) >>: simple coercion.

<:expr< (e : $t1$:> $t2$) >>: coercion with type constraint.

<:expr< $flo:s$ >>: float constant; "s" is a string representing the float.

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

<:expr< fun [$list:pwel$] >>: function; the parameter "pwel" is a list of (patt * option expr * expr), the optional expr being the optional when construction.

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

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

<:expr< ~ i : e >>: label.

<:expr< lazy e >>: lazy.

<:expr< let $opt:b$ $list:pel$ in e >>: let statement (b = recursive flag)

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

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

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

<:expr< do { $list:el$ } >>: sequence.

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

<:expr< $str:s$ >>: (escaped) string constant.

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

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

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

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

<:expr< while e do { $list:el$ } >>: while statement.

A.3 Quotation `patt`

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

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

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

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

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

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

`<:patt< $chr:c$ >>`: (escaped) character constant.

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

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

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

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

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

`<:patt< $str:s$ >>`: (escaped) string constant.

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

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

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

A.4 Important remark about lists

In the syntax tree, there is no node directly representing lists. Therefore, if “`el`” is a list of values of type `MLast.expr`, these codes DO NOT create the node of their list:

```
<:expr< $list:el$ >>       (* does not work! *)
<:expr< [ $list:el$ ] >>   (* does not work either! *)
```

The only available nodes are: the “`cons`” constructor (“`::`”) and the “`nil`” constructor (“`[]`”). To create a a list node, you have therefore to apply them to your list elements, one by one. This can be done by this code:

```
List.fold_right (fun x l -> <:expr< [$x$ :: $l$] >>) el <:expr< [] >>
```

Notice that it is different with arrays. This code:

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

actually builds an array whose elements are given in the expression list.

A.5 Quotation `ctyp`

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

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

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

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

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

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

`<:ctyp< # $list:sl$ >>`: class type.

`<:ctyp< ~ $s$ : $t$ >>`: label type.

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

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

`<:ctyp< < $list:fl$ > >>`: object type.

`<:ctyp< ? $s$ : $t$ >>`: optional label type.

`<:ctyp< ! $list:sl$ . $t$ >>`: polymorphic type.

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

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

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

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

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

`<:ctyp< [| $list:rfl$ |] >>`: variant type definition.

A.6 Quotation `sig_item`

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

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

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

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

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

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

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

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

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

A.7 Quotation str_item

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

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

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

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

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

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

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

<:str_item< open sl >>: open.

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

<:str_item< value $opt:b$ $list:pel$ >>: variables definition (b = recursive flag).

A.8 Quotation module_type

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

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

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

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

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

<:module_type< 'i >>: module type variable (abstract).

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

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

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

A.9 Quotation module_expr

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

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

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

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

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

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

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

A.10 Quotation `class_expr`

Syntax tree nodes for language classes, of type `MLast.class_expr`. The basic nodes are:

`<:class_expr< $ce$ $e$ >>`: application of a class to an expression.

`<:class_expr< $list:sl$ [ $list:tl$ ] >>`: class constructor with its parameters (`sl` is a string list representing its access list, and `tl` a type list). If no parameter, the `[ ]` part can be omitted.

`<:class_expr< fun $p$ -> $ce$ >>`: class function

`<:class_expr< let $opt:b$ $list:lb$ in $ce$ >>`: let binding (b = recursive flag)

`<:class_expr< object $opt:s$ $list:csil$ end >>`: object (s = optional self)

`<:class_expr< ( $ce$ : $ct$ ) >>`: type constraint

A.11 Quotation `class_type`

Syntax tree nodes for language classes types, of type `MLast.class_type`. The basic nodes are:

`<:class_type< $list:sl$ [ $list:tl$ ] >>`: class constructor with its parameters (`sl` is a string list representing its access list, and `tl` a type list). If no parameter, the `[ ]` part can be omitted.

`<:class_type< [ $t$ ] -> $ct$ >>`: arrow class type

`<:class_type< object $opt:s$ $list:csil$ end >>`: class object type (s = optional self)

A.12 Quotation `class_sig_item`

Syntax tree nodes for language classes signatures (object types) items, of type `MLast.class_sig_item`. The basic nodes are:

`<:class_sig_item< type $t1$ = $t2$ >>`: constraint.

`<:class_sig_item< declare $list:csil$ end >>`: declare.

`<:class_sig_item< inherit $ct$ >>`: inheritance.

`<:class_sig_item< method $opt:pr$ $s$ : $t$ >>`: method type (pr = private flag)

`<:class_sig_item< method virtual $opt:pr$ $s$ : $t$ >>`: virtual method type (pr = private flag).

`<:class_sig_item< value $opt:m$ $s$ = $t$ >>`: value (m = mutable flag)

A.13 Quotation class_str_item

Syntax tree nodes for language classes structures (objects) item, of type MLast.class_str_item. The basic nodes are:

<:class_str_item< declare $list:csil$ end >>: declare.

<:class_str_item< type $t1$ = $t2$ >>: constraint.

<:class_str_item< inherit ct >>: inheritance.

<:class_str_item< initializer e >>: initialization.

<:class_str_item< method $opt:pr$ s : t >>: method (pr = private flag).

<:class_str_item< method virtual $opt:pr$ s : t >>: virtual method (pr = private flag)

<:class_str_item< value $opt:m$ s = t >>: value (m = mutable flag).