Stagiaire Nicolas Pouillard
Spécialisation CSI
Promotion 2006

Annexes : Rénovation de Camlp4, un Préprocesseur Pretty-Printeur pour Caml

Ces documents sont rédigés en anglais car ils ont été diffusés aux utilisateurs de Camlp4.

Table des matières

Chapitre 1

Nouvelle architecture

- `Camlp4` :
  - `PreCast` : This module is a good start for most of Camlp4 users.
  - `Sig` : The Sig module contains all signatures used by Camlp4. It's like an interface repository. You can use this module to learn what is possible to do with a module.

    List of useful signatures :
    - `Loc` : Abstract *location* values.
    - `Camlp4Ast` : The OCaml abstract syntax tree together with mapping class and functions.
    - `Quotation` : Quotations and *quotation* expanders.
    - `Token`, `Camlp4Token` : Abstract tokens, and OCaml tokens.
    - `Lexer` : The signature of lexing functions.
    - `Grammar.Static,Grammar.Dynamic` : The signature of grammars and entries manipulation.
    - `Syntax`, `Camlp4Syntax` : The syntax signature contains most of needed things in order to make an extension. The Camlp4Syntax signature is more OCaml specific. (see also AntiquotSyntax, SyntaxExtension)
    - `Parser`, `Printer` : Minimal parser and printer signatures.
    - `AstFilters` : The signature for registering and call Ast filters.
    - Misc signatures : `Ast`, `Warning`, `Error`, `DynLoader`, `Type`, `Id`, `Options`
  - `Struct` : Implementations of these signatures. Only useful when you want to build a customized Camlp4 and reuse modules. (For Jedis only :)).
  - `Register` : This module allows to register syntax extensions, parsers, printers, and filters.
  - `Config`, `ErrorHandler` and `Debug` : at top level for visibility purpose.

- `Camlp4Parsers` :
  - `OCaml` : The original OCaml syntax.
  - `OCamlr` : The revised OCaml syntax.
  - `OCamlRevisedQuotation` : OCaml *quotations* in the revised syntax.
  - `OCamlParser`, `OCamlRevisedParser` : The parser and stream syntax.
  - `Grammar` : The syntax of extensible grammars (`EXTEND...END`).
  - `Macro` : Macros like `IFDEF`, `DEFINE`, `INCLUDE`, `__FILE__`...
  - `Debug` : A debug keyword for auto-removable debugging info.
  - `LoadCamlp4Ast` : Load an `Camlp4Ast` marshalled value.
- `Camlp4Printers` :
  - `OCaml` : A re-write from scratch OCaml original syntax *pretty-printer*.
  - `OCamlr` : Revised *pretty-printer*.
  - `DumpOCamlAst`, `DumpCamlp4Ast` : Marshalling functions of trees for the OCaml compiler and CAMLP4 itself.
- `Camlp4Filters` :
  - `ExceptionTracer` : Add "try with" blocks around the body of functions that prints (with the *location*) and re-raise any exception (useful for debugging).
  - `GenerateMap` : Grab type definitions and generate map functions to traverse these structures.
  - `LiftCamlp4Ast` : Associate to a tree the program that produces it. (f 42 ==¿ Ast.ExApp loc (Ast.ExLid loc "f") (Ast.ExInt loc "42") == <:*expr*< `f` 42 >>)
  - `StripLocations` : Replace all *location* information by the ghost *location*.
- `Camlp4Top` :
  - `Camlp4Top` : The main top-level plugin for the OCaml top-level.
  - `Rprint` : The revised *pretty-printer* for values.

Chapitre 2

La procédure de *bootstrap*

When one makes a change in CAMLP4 sources, one must understand how proceeds the *bootstrap* process. Generally each major change must be split in two parts. The first one is to build a CAMLP4 that understands the needed features, the second is to update the CAMLP4 in order to use these features. So you can change the grammars specification, the pretty-printers and the code expansion but not directly use these changes.

So to compile `camlp4` just type make in the `camlp4` directory.

To make this process more realistic one can take a simple example. Suppose that one want to change the EXTEND Gram ... END syntax. This syntax produce function calls like Gram.extend ... The proposed example is to change the name of this function to broaden.

For this trivial example only one working copy of CAMLP4 suffices. However in case of a large change my advice is to use two working copies, one for parsing, printing and code generation changes and the other for code change. The two working copies system is needed in order to keep the compilation time reasonable when multiple changes are performed on the first one.

2.1 Edit the appropriate file : Camlp4Parsers/Grammar.ml

Find the `.extend` sequence and replace it by `.broaden` you can check that this change won't affect your running CAMLP4 since it occurs inside a *quotation*.

2.2 Compile this new CAMLP4

```
$ make
```

In general one change parsers, printers, and code generation until satisfied by what can parse, or generate this version.

2.3 Test it

```
$ cat test.ml
```

EXTEND *Gram expr*: [[]]; **END**

```
$ ./camlp4of.run pr_o.cmo test.ml
```

let _ = *Gram.extend*' (*expr* : '*expr Gram.Entry.t*) (*None*, [*None*, *None*, []])

2.4 Propagate the change in the whole Camlp4

Now one should change the `extend` definition and signature to rename it `broaden`. In this case this involves the following files :

- Camlp4/Sig/Grammar/Dynamic.mli : The signature for dynamic grammars
- Camlp4/Sig/Grammar/Static.mli : The signature for static grammars
- Camlp4/Struct/Grammar/Static.ml : The structure for dynamic grammars
- Camlp4/Struct/Grammar/Dynamic.ml : The structure for static grammars
- Camlp4/Struct/Grammar/Insert.ml : The real implementation

2.5 Bootstrap it

- With one copy :
  ```
  $ make bootstrap
  ```
- With two copies :
  ```
  $ cp camlp4boot.run /.../second_copy/camlp4/boot/camlp4boot
  $ cd /.../second_copy/camlp4
  $ make clean all
  ```

2.6 In case of *bootstrap* failure

- With one copy (rebuild the previous Camlp4) :
  ```
  $ make restore
  $ make
  $ edit...
  ```

```
$ make bootstrap
```

– With two copies :

```
$ cd /.../first_copy/camlp4
$ edit...
$ make
$ cp camlp4boot.run /.../second_copy/camlp4/boot/camlp4boot
$ cd /.../second_copy/camlp4
$ make
```

2.7 Reach the fix-point

When your first *bootstrap* reaches the end you must be sure that you reached a fix-point so you must re-run the *bootstrap* process (with only one copy, the second) until the message "Fixpoint reached, *bootstrap* succeeded."

Chapitre 3

Changements dans les extensions de syntaxe

3.1 Before

Let two grammar extensions *in the old syntax* `dynamic_old_syntax.ml` and `static_old_syntax.ml`.

```
(* dynamic_old_syntax.ml *)
type t1 = A | B
type t2 = Foo of string * t1
open Pcaml
let foo = Entry.mk gram "foo"
let bar = Entry.mk gram "bar"
EXTEND
  GLOBAL: foo bar;
  foo: [ [ "foo"; i = LIDENT; b = bar -> Foo(i, b) ] ];
  bar: [ [ "?" -> A | "." -> B ] ];
END;;
Entry.parse foo (Stream.of_string "foo␣x?") = Foo("x", A)
DELETE_RULE foo: "foo"; LIDENT; bar END
```

```
(* static_old_syntax.ml *)
type t1 = A | B
type t2 = Foo of string * t1
module Gram = Grammar.GMake(...)
let foo = Gram.Entry.mk "foo"
let bar = Gram.Entry.mk "bar"
GEXTEND Gram
  GLOBAL: foo bor;
  foo: [ [ "foo"; i = LIDENT; b = bar -> Foo(i, b) ] ];
  bar: [ [ "?" -> A | "." -> B ] ];
```

```
END;;
Gram.Entry.parse foo (Stream.of_string "foo␣x?") = Foo("x", A)
GDELETE_RULE Gram foo: "foo"; LIDENT; bar END
```

3.2 The quick and non extensible way : the `Camlp4.PreCast` module

```
  (* quick_non_extensible_example.ml *)
(* This scheme only works when the grammar value is not really
   used for other things than entry creation. In fact grammars
   are now static by default. *)
type t1 = A | B
type t2 = Foo of string * t1
open Camlp4.PreCast
open Syntax
let foo = Gram.Entry.mk "foo"
let bar = Gram.Entry.mk "bar"
EXTEND Gram
  GLOBAL: foo bar;
  foo: [ [ "foo"; i = LIDENT; b = bar -> Foo(i, b) ] ];
  bar: [ [ "?" -> A | "." -> B ] ];
END;;
Gram.parse_string foo (Loc.mk "<string>") "foo␣x?" = Foo("x", A)
DELETE_RULE Gram foo: "foo"; LIDENT; bar END
```

3.3 The functorial way

```
  (* dynamic_functor_example.ml *)
type t1 = A | B
type t2 = Foo of string * t1
open Camlp4

module Id = struct (* Information for dynamic loading *)
  let name = "My_extension"
  let version = "$Id$"
end

(* An extension is just a functor: Syntax -> Syntax *)
module Make (Syntax : Sig.Syntax.S) = struct
  include Syntax
  let foo = Gram.Entry.mk "foo"
  let bar = Gram.Entry.mk "bar"
  open Camlp4.Sig.Camlp4Token
  EXTEND Gram
    GLOBAL: foo bar;
    foo: [ [ "foo"; i = LIDENT; b = bar -> Foo(i, b) ] ];
    bar: [ [ "?" -> A | "." -> B ] ];
  END;;
  Gram.parse_string foo (Loc.mk "<string>") "foo␣x?" = Foo("x", A)
  DELETE_RULE Gram foo: "foo"; LIDENT; bar END
end

(* Register it to make it usable via the camlp4 binary. *)
module M = Register.SyntaxExtension(Id)(Make)
```

```
(* static_functor_example.ml *)
```

```
type t1 = A | B
type t2 = Foo of string * t1
open Camlp4.PreCast
module Lexer = struct
  ... if you need a different lexer ...
end
module Gram = MakeGram(Lexer)
let foo = Gram.Entry.mk "foo"
let foo = Gram.Entry.mk "foo"
EXTEND Gram
  GLOBAL: foo;
  foo: [ [ "foo"; i = LIDENT; b = bar -> Foo(i, b) ] ];
  bar: [ [ "?" -> A | "." -> B ] ];
END;;
Gram.parse_string foo (Loc.mk "<string>") "foo␣x?" = Foo("x", A)
DELETE_RULE Gram foo: "foo"; LIDENT; bar END
```