Syntax Extensions

OCaml PRO – Grégoire Henry

March 9-13 2015

Outline

History
Extension nodes & annotations
PPX
MetaOCaml

History

The past: Camlp4

Camlp4: a Pre-Processor and a Pretty-Printer.

- introduced in the late '90;
- an original extensible parser technology;
  - easy to embed domain specific language into OCaml;
  - extend the OCaml syntax with syntactic-sugar;
  - much slower than classical yacc-generated parser.
- a endless source of social conflict:
  - a burben for mainteners: there were ~~two~~ three official OCaml parsers;
  - syntax-extension: *"Is this really OCaml code? My IDE do not agree."*
  - camlp4 vs. camlp5: two incompatible versions maintened since 2007.

Starting with ocaml-4.02, Camlp4 has been removed from the official distribution.

The present: PPX & extension nodes

Camlp4 has been replaced with two complementary mechanisms:

PPX Pre-Processor-eXtension.

- a new compiler option `-ppx`
- the syntax tree may be piped through an external program

Extension nodes & annotations

- Annotations: decorations of the syntax tree
- Extension nodes: generic placeholders in the syntax tree
- Properly quoted strings

Extension nodes & annotations

Examples

Annotations

```
type t = A | B
  [@@deprecated "Please␣use␣type␣'s'␣instead."]

let f x =
  assert (x >= 0) [@ppwarning "TODO:␣remove␣this␣later"];
  (* ... *)
```

Extensions nodes

```
let printer : (int * int) list -> string =
  [%show: (int * int) list]
let user = [%getenv "USER"]
let f x = match%regexp x with
  | "a.*b" -> true
  | "a+b+" -> true
  | _ -> false
```

- composed of an identifier and a payload that is valid OCaml code:

$$\begin{array}{lcl} \textit{attribute} & ::= & [@\ \textit{id}\ \textit{struct\_item}] \\ & | & [@\ \textit{id}\ ?\ \textit{pattern}] \\ & | & [@\ \textit{id} : \textit{type\_expr}] \end{array}$$

- three kinds of possible *attachment*:

`[@ ...]` the previous expression, type expression, pattern, ...

`[@@ ...]` the previous structure item, or signature item:

  - a type definition
  - a value definition
  - an exception definition
  - ...

`[@@@ ...]` floating annotations, between structure items or signature items.

- alternatives syntaxes:

```
let [@foo][@bar x] x = 2 in x + 1 === (let x = 2 in x + 1
begin[@foo] ... end
=== (begin ... end)[@foo]
```

Annotations

Predefined annotations:

warning allow to *locally* change the CLI option `-w` (warnings to be reported)
warnerror allow to *locally* change the CLI option `-warn-error`;
deprecated when the element is later referenced, a warning (3) is triggered;
ppwarning the string payload is reported as warning (22) by the compiler;
error the string payload is reported as an error by the compiler.

```
module X = struct
  ... (* 12 enabled, 9 not enabled *)
  [@@@warning "+9"]
  ... (* both enabled *)
end [@@warning "+12"]
(* none enabled *)
```

Extension nodes

- composed of an identifier and a payload that is valid OCaml code:

$$\begin{array}{lcl} \textit{attribute} & ::= & [\%\ \textit{id}\ \textit{struct\_item}] \\ & | & [\%\ \textit{id}\ ?\ \textit{pattern}] \\ & | & [\%\ \textit{id} : \textit{type\_expr}] \end{array}$$

- two kinds of possible *positioning*:

  `[% ...]` inside an expression, type expression, pattern, ...

  `[%% ...]` the previous structure item, or signature item:

    - a type definition
    - a value definition
    - an exception definition
    - ...

- alternatives syntaxes:

```
let%foo x = 2 in x+1      === [%foo let x = 2 in x+1]
begin%foo ... end         === [%foo begin ... end]
begin%foo[@bar] ... end   === [%foo (...) [@bar]]
```

- extensions nodes are considered as error by the type checker.

Quoted strings

```
let data = "␣Usual␣strings:␣\"Hello␣world!\\n\"␣"
let data = {| Usual strings: "Hello␣world!\n" |}
let html = {|html|
  <html lang="en"> ...  </html>
|html|}
```

PPX

ppx_deriving[a] by Peter Zotov, allows to derive function from type and type definitions.

```
# type s = A | B of int [@@deriving show];;
type t = A | B of i
val pp_s : Format.formatter -> s -> unit = <fun>
val show_s : s -> bytes = <fun>
# show (B 1);;
- : bytes = "B 1"
# [%deriving.show: (int * s) list] [1, A; 3, B 4];;
- : bytes = "[(1, A); (3, B 4)]"
```

Existing derivers: show, eq, ord, enum, iter, map, fold, create, yojson, protobuf.

[a] https://github.com/whitequark/ppx_deriving

Real PPX examples

sedlex[a] by Alain Frisch (Lexifi), is Unicode-friendly lexer generator for OCaml.

```
let digit = [%sedlex.regexp? '0'..'9']
let number = [%sedlex.regexp? Plus digit]
let rec token buf =
  let letter = [%sedlex.regexp? 'a'..'z'|'A'..'Z'] in
  match%sedlex buf with
  | number ->
    Printf.printf "Number %s\n" (Latin1.lexeme buf); token bu
  | letter, Star ('A'..'Z' | 'a'..'z' | digit) ->
    Printf.printf "Ident %s\n" (Latin1.lexeme buf); token buf
  | Plus xml_blank -> token buf
  | Plus (Chars "+*-/") ->
    Printf.printf "Op %s\n" (Latin1.lexeme buf); token buf
  | 128 .. 255 -> print_endline "Non ASCII"
  | eof -> print_endline "EOF"
  | _ -> failwith "Unexpected character"
```

[a] https://github.com/alainfrisch/sedlex/

How to write PPX?

- Explore the OCaml syntax tree in the OCaml compiler sources:
    - `parsing/parsetree.mli`
    - `parsing/asttypes.mli`
- Explore the helpers functions provided in:
    - `parsing/ast_helpers.mli`
    - `parsing/ast_mapper.mli`
- Minimalistic example:

```ocaml
let test_mapper argv =
  { default_mapper with
    expr = fun mapper expr ->
      match expr with
      | { pexp_desc =
          Pexp_extension ({ txt = "test" }, PStr [])} ->
        Exp.constant (Const_int 42)
      | other -> default_mapper.expr mapper other; }

let () = register "ppx_test" test_mapper
```

How to write PPX? 2 / 3

ppx_tools[a] by Alain Frisch (Lexifi), is a set of tools for authors of syntactic tools.

dumpast print the syntax as valid OCaml pattern

```
# ocamlfind ppx_tools/dumpast -e "1 + 2"
{pexp_desc =
  Pexp_apply ({pexp_desc = Pexp_ident {txt = Lident "+"}},
   [("", {pexp_desc = Pexp_constant (Const_int 1)});
    ("", {pexp_desc = Pexp_constant (Const_int 2)})])}
```

rewriter pretty-printer for a rewrited source file

```
# ocamlfind ppx_tools/rewriter ./my_ppx sample.ml
```

ppx_metaquot a ppx for easiest matching and generation of OCaml syntax tree.

[a] https://github.com/alainfrisch/ppx_tools/

How to write PPX?


```
let test_mapper argv =
  { default_mapper with
    expr = fun mapper expr ->
      match expr with
      | [%expr [%test] ] -> [%expr 42]
      | other -> default_mapper.expr mapper other; }

let () = register "ppx_test" test_mapper
```

MetaOCaml

- PPX allows to generate code. But it does help to generate valid or well-typed code.
- BER MetaOCaml is a conservative extension of OCaml for “writing programs that generate programs”.
- A well-typed BER MetaOCaml program generates only well-scoped and well-typed programs: The generated code shall compile without type errors.

```
let square x = x * x
let rec power n x =
  if n = 0 then 1
  else if n mod 2 = 0 then square (power (n/2) x)
  else x * (power (n-1) x)
(* val power : int -> int -> int = <fun> *)
```

```
let rec spower n x =
  if n = 0 then .<1>.
  else if n mod 2 = 0 then .< square .~(spower (n/2) x) >.
  else .< .~x * .~(spower (n-1) x) >.;;
(* val spower : int -> int code -> int code = <fun> *)
let spower7_code = .<fun x -> .~(spower 7 .<x>.)>.;;
let spower7 = !. spower7_code
(* val spower7 : int -> int = <fun> *)
```

Further readings

Extension points and PPX

- `http://www.lexifi.com/blog/ppx-and-extension-points`
- `http://caml.inria.fr/pub/docs/manual-ocaml/extn.html#sec241`
- `http://caml.inria.fr/pub/docs/manual-ocaml/extn.html#sec243`

BER MetaOCaml

- `http://okmij.org/ftp/ML/MetaOCaml.html`

Camlp4

- `http://www.mauny.net/data/papers/mauny-de-rauglaudre-1996.pdf`
- `https://github.com/ocaml/camlp4/wiki`
- `http://camlp5.gforge.inria.fr/`