Features of Camlp4

CS 4215: Programming Language Implementation

Chin Wei Ngan

March 31, 2015

Generated on Tuesday 31st March, 2015, 11:52

Outline

1. Parsers and Grammars
2. Grammars
3. OCaml AST
4. Camlp4 Quotation
5. Consuming OCaml AST
6. Filter
7. Lexer
8. Syntax Extension

Stream and Parsers

- Syntax extension of camlp4 achieved primarily via streams and parsers.
- A stream is a value of abstract type: `'a Stream.t`.
  An example of stream is `[<'3;'1;'4;'5>]`.
- A parser is a function of type: `'a Stream.t -> 'b`
  An example of a parser is `parser [< 'x >] -> x`.
- A parser may return three possible outcomes:
  1. Successful outcome
  2. Exception `Stream.Failure` which indicates that no first element matches the pattern
  3. Exception `Stream.Error` which indicates that first element match input stream, but not the rest.

Printing Translation in OCaml Format

```
let s = [<'3;'1;'4;'5>]

let p = parser [< 'x >] -> x
```

Code Printing Command

`camlp4o ex6a_stream_parser.ml -printer o`

```
let s = Stream.icons 3 (Stream.icons 1
          (Stream.icons 4 (Stream.ising 5)))

let p (__strm : _ Stream.t) =
  match Stream.peek __strm with
  | Some x -> (Stream.junk __strm; x)
  | _ -> raise Stream.Failure
```

Printing OCaml in Revised Syntax

```
let s = [<'3;'1;'4;'5>]

let p = parser [< 'x >] -> x
```

Code Printing Command

```
camlp4o ex6a_stream_parser.ml -printer r
```

```
value s = Stream.icons 3 (Stream.icons 1 (Stream.icons 4 (Stream.ising
value p (__strm : Stream.t _) =
  match Stream.peek __strm with
  [ Some x -> (Stream.junk __strm; x)
  | _ -> raise Stream.Failure ];
```

Rationale for Revised Syntax

- OCaml quotation not well-supported in the original syntax.
- Some AST construction are difficult or impossible to create using original syntax.
- Can specify which original or revised syntax on a per file basis.
- Information on revised syntax:

```
http://caml.inria.fr/pub/docs/tutorial-camlp4 \
  /tutorial005.html
```

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

Different versions of Camlp4 executables

Easily distinguished by the modules loaded.

```
> camlp4 -loaded-modules
```

```
> camlp4r -loaded-modules
Camlp4.Printers.OCaml
Camlp4OCamlRevisedParser
Camlp4OCamlRevisedParserParser
```

```
> camlp4r -loaded-modules
Camlp4.Printers.OCaml
Camlp4OCamlParser
Camlp4OCamlParserParser
Camlp4OCamlRevisedParser
Camlp4OCamlRevisedParserParser
```

Full version Camlp4 executable

Supports grammar form, quotation expanders and more.

```
> camlp4of -loaded-modules
Camlp4.Printers.OCaml
Camlp4GrammarParser
Camlp4ListComprehension
Camlp4MacroParser
Camlp4OCamlParser
Camlp4OCamlParserParser
Camlp4OCamlRevisedParser
Camlp4OCamlRevisedParserParser
Camlp4QuotationExpander
```

StandAlone vs Pre-processor Modes

- Standalone mode pretty-print code in OCaml regular syntax.
  `camlp4o ex6a_stream_parser.ml`
- Pre-processor mode outputs code in an AST format understood by OCaml compiler via `Camlp4Printers.DumpOCamlAst`

  ```
  ocamlc -pp camlp4o -c ex6a_stream_parser.ml
  ```

  yields `ex6a_stream_parser.cmo`
- Pre-processing, compiling and linking with camlp4:

  ```
  ocamlc camlp4o -o ex6a \
    -c ex6a_stream_parser.ml
  ```

  yields bytecode executable `ex6a`

A Recursive Parser

A concise way of writing parsers using recursion and streams.

An example : `ex6_foo.ml`

```
let rec p = parser
  | [< '"foo"; 'x; '"bar" >] -> "foo-bar+" ^ x
  | [< '"baz"; y = p >] -> "baz+" ^ y
```

Translation for Recursive Parser

Translation : `camlp4o ex6_foo.ml`

```
let rec p (__strm : _ Stream.t) =
  match Stream.peek __strm with
  | Some "foo" ->
    (Stream.junk __strm;
     (match Stream.peek __strm with
      | Some x ->
         (Stream.junk __strm;
         (match Stream.peek __strm with
           | Some "bar" -> (Stream.junk __strm; "foo-bar+" ^ x)
           | _ -> raise (Stream.Error "")))
      | _ -> raise (Stream.Error "")))
  | Some "baz" ->
     (Stream.junk __strm;
      let y =
        (try p __strm
         with | Stream.Failure -> raise (Stream.Error ""))
      in "baz+" ^ y)
  | _ -> raise Stream.Failure
```

Limited Backtracking

One lookahead allows another parser to be called if the first element of prior parser fails.

Code : `ex7_backtrack.ml`

```
let rec p = parser
  | [< x = q >] -> x
  | [< '"bar" >] -> "bar"
```

Translation for Parser with Backtracking

Translation : `camlp4o ex7_backtrack.ml`

```
let rec p (__strm : _ Stream.t) =
  try q __strm
  with
  | Stream.Failure ->
      (match Stream.peek __strm with
       | Some "bar" -> (Stream.junk __strm; "bar")
       | _ -> raise Stream.Failure)
```

Using Grammar Rules

- Grammar rules provide a more succinct way to write parsers.
- It uses lexer to tokenize words
- It provides a hierarchical way to name non-terminal grammar entries.
- It supports grammar rules and action/result.
- The grammar rules are extensible.

Example Grammar Module

File : `ex8_grammar_foo.ml`

```
open Camlp4.PreCast
module Gram = MakeGram(Lexer)
let expr = Gram.Entry.mk "expr"
EXTEND Gram
  expr:
    [[
       "foo"; x = LIDENT; "bar" -> "foo-bar+" ^ x
     | "baz"; y = expr -> "baz+" ^ y
     ]];
END;;

try
  print_endline
    (Gram.parse_string expr Loc.ghost Sys.argv.(1))
with Loc.Exc_located (_, x) -> raise x
```

Translation for Grammar Module

Translation : `camlp4of ex8_grammar_foo.ml`

```
open Camlp4.PreCast
module Gram = MakeGram(Lexer)
let expr = Gram.Entry.mk "expr"
let _ =
  Gram.extend (expr : 'expr Gram.Entry.t)
   ((fun () ->
    (None,
     [ (None, None,
       [ ([ Gram.Skeyword "baz"; Gram.Sself ],
         (Gram.Action.mk
           (fun (y : 'expr) _ (_loc : Gram.Loc.t) ->
             ("baz+" ^ y : 'expr))));
....

let _ =
  try print_endline (Gram.parse_string expr
    Loc.ghost Sys.argv.(1))
  with | Loc.Exc_located (_, x) -> raise x
```

Remarks on Grammar Code

- Grammar module from `Camlp4.PreCast` is an empty grammar with a default lexer.
- `expr` is a non-terminal grammar entry newly created.
- `Gram.parse_string` can parse a string using an entry.
- `LIDENT` is a lower-case identifier of the default lexer.
- The grammar rule may be recursive.

Translation, Compiling and Linking

- Translation :

```
camlp4of ex8_grammar_foo.ml
```

- Compiling with Pre-processor :

```
ocamlc -I +camlp4 -pp camlp4of \
 -c ex8_grammar_foo.ml
```

- Compiling with Pre-processor and linking :

```
ocamlc -I +camlp4 -pp camlp4of -o ex8 \
 dynlink.cma camlp4lib.cma ex8_grammar_foo.ml
```

Failed BackTracking

Due to shallow lookahead, some backtracking of parsing may fail. An example is:

```
EXTEND Gram
  GLOBAL: expr;
  g: [[ "plugh" ]];
  f1: [[ g; "quux" ]];
  f2: [[ g; "xyzzy" ]];
  expr:
    [[ f1 -> "f1" | f2 -> "f2" ]];
END;;
```

Input: `plugh quux  ---> f1`
Input: `plugh xyzzy ---> EXCEPTION`

Adding Lookahead Test

Adding a 2-token lookahead test:

```
let test =
 Gram.Entry.of_parser "test" (fun strm ->
   match Stream.npeek 2 strm with
    | [ _; KEYWORD "xyzzy", _ ] -> raise Stream.Failure
    | _ -> ())
EXTEND Gram
  ...
  expr:
    [[ test; f1 -> "f1" | f2 -> "f2" ]];
END;;
```

Input: `plugh quux  ---> f1`
Input: `plugh xyzzy ---> f2`

AST for OCaml Code

- This is quite complex but essential for syntax extension.
- Camlp4 provides a quotation mechanism to make it easier to write and manipulate GST.
- Quotation allows concrete syntax to be written.
- Antiquotation allows AST to be spliced back into some quotation.

Example of Quotation and AST

File : `ex2_ast.ml`

```
let q = <:str_item< let f x = x >>
```

Printing AST : `camlp4of ex2_ast.ml -printer o`

```
let q =
 Ast.StVal (_loc, Ast.ReNil,
  (Ast.BiEq (_loc, (Ast.PaId (_loc, (Ast.IdLid (_loc, "f")))),
   (Ast.ExFun (_loc,
    (Ast.McArr (_loc, (Ast.PaId (_loc, (Ast.IdLid (_loc, "x")))),
     (Ast.ExNil _loc), (Ast.ExId (_loc, (Ast.IdLid (_loc, "x"))))
               )))))))
```

Original and Revised Syntax

Original Syntax : `camlp4of ex2_ast.ml -printer o`

```
let q =
 Ast.StVal (_loc, Ast.ReNil,
  (Ast.BiEq (_loc, (Ast.PaId (_loc, (Ast.IdLid (_loc, "f")))),
   (Ast.ExFun (_loc,
    (Ast.McArr (_loc, (Ast.PaId (_loc, (Ast.IdLid (_loc, "x")))),
     (Ast.ExNil _loc), (Ast.ExId (_loc, (Ast.IdLid (_loc, "x"))))
              )))))))
```

Revised Syntax : `camlp4of ex2_ast.ml -printer r`

```
value q =
 Ast.StVal _loc Ast.ReNil
  (Ast.BiEq _loc (Ast.PaId _loc (Ast.IdLid _loc "f"))
   (Ast.ExFun _loc
    (Ast.McArr _loc (Ast.PaId _loc (Ast.IdLid _loc "x"))
     (Ast.ExNil _loc) (Ast.ExId _loc (Ast.IdLid _loc "x")))));
```

AST for OCaml Type

Example Type AST : `ex2a_type_ast.ml`

```
let q = <:ctyp< ('a, 'b) foo >>
let r = <:ctyp< 'a 'b foo >>
```

`camlp4of ex2a_type_ast.ml -printer r`

```
value q =
 Ast.TyApp _loc
  (Ast.TyApp _loc (Ast.TyId _loc
    (Ast.IdLid _loc "foo")) (Ast.TyQuo _loc "a"))
  (Ast.TyQuo _loc "b");
value r =
 Ast.TyApp _loc (Ast.TyId _loc (Ast.IdLid _loc "foo"))
  (Ast.TyApp _loc (Ast.TyQuo _loc "b") (Ast.TyQuo _loc "a"));
```

OCaml AST a bit strange. Type application used in different ways.

Type Generation Example

Let us generate an enumerated type.

Code `ex3_data_gen.ml`

```
open Camlp4.PreCast
let _loc = Loc.ghost ;;

let cons = ["Foo";"Bar";"Baz"];;

Printers.OCaml.print_implem
<:str_item<
 type t =
  $ Ast.TySum (_loc,
     Ast.tyOr_of_list
      (List.map
        (fun c -> <:ctyp< $uid:c$ >>)
          cons)) $
>>;;
```

Source of camlp4 methods in:

`git clone https://github.com/ocaml/camlp4.git`

Compiling, Linking and Execution

```
ocamlc -I +camlp4 -pp camlp4of -o ex3 \
  dynlink.cma camlp4lib.cma ex3_data_gen.ml
```

This produced executable `ex3`. Executing it gives output:

```
type t = | Foo | Bar | Baz;;
```

Code Generation

Code to Generate a Printer:

```
let to_string = function
  $Ast.mcOr_of_list
   (List.map (fun c ->
     <:match_case< $uid:c$ -> $`str:c$ >>)
        cons)$
```

Generated Printer:

```
let to_string = function | Foo -> "Foo"
  | Bar -> "Bar" | Baz -> "Baz";;
```

Code Generation

Code to Generate a Parser for Type:

```
let of_string = function
  $let ors =
    Ast.mcOr_of_list
     (List.map (fun c ->
       <:match_case< $‘str:c$ -> $uid:c$ >>)
          cons) in
    Ast.McOr(_loc,ors,
      <:match_case< _ -> invalid_arg "bad string" >>)$
```

Generated Parser for Type:

```
let of_string =
  function
  | "Foo" -> Foo
  | "Bar" -> Bar
  | "Baz" -> Baz
  | _ -> invalid_arg "bad string";;
```

1. Parsers and Grammars
2. Grammars
3. OCaml AST
4. Camlp4 Quotation
5. Consuming OCaml AST
6. Filter
7. Lexer
8. Syntax Extension

Otags - Code Consumption

Minimal Broken File : `ex4_otags.ml`

```
open Camlp4.PreCast
module M = Camlp4OCamlRevisedParser.Make(Syntax)
module N = Camlp4OCamlParser.Make(Syntax)

let files = ref []

let rec do_fn fn =
 let st = Stream.of_channel (open_in fn) in
 let str_item = Syntax.parse_implem (Loc.mk fn) st in
 let str_items = Ast.list_of_str_item str_item [] in
 let tags = List.fold_right do_str_item str_items [] in
 files := (fn, tags)::!files
```

We parse OCaml code and convert to a single module `str_item`.

We extract a list of definitions before computing tags for each.

Otags - Code Consumption

Code locates bindings before gathering the type tags.

```
and do_str_item si tags =
  match si with
 (* | <:str_item< let $rec:_$ $bindings$ >> -> *)
    | Ast.StVal (_, _, bindings) ->
        let bindings = Ast.list_of_binding bindings [] in
        List.fold_right do_binding bindings tags
    | _ -> tags

and do_binding bi tags =
 match bi with
  | <:binding@loc< $lid:lid$ = $_$ >> ->
    let line = Loc.start_line loc in
    let off = Loc.start_off loc in
    let pre = "let " ^ lid in
    (pre, lid, line, off)::tags
  | _ -> tags
```

Otags - Code Consumption

Printer for generating tag files.

```
let print_tags files =
  let ch = open_out "TAGS" in
  ListLabels.iter files ~f:(fun (fn, tags) ->
    Printf.fprintf ch "\012\n%s,%d\n" fn 0;
    ListLabels.iter tags ~f:(fun (pre, tag, line, off) ->
      Printf.fprintf ch "%s\127%s\001%d,%d\n" pre tag line off))

;;
Arg.parse [] do_fn "otags: fn1 [fn2 ...]";
print_tags !files
```

Compiling executable:

```
ocamlc -I +camlp4 -I +camlp4/Camlp4Parsers -pp camlp4of \
 -o ex4 dynlink.cma camlp4fulllib.cma ex4_otags.ml
```

Transforming OCaml AST witn Filters

- camlp4 best used to pre-process OCaml code
- OCaml compiler can parse code
- Filter can transform AST

An Example Filter

To hook into Camlp4 plugin mechanism, we define filter as a functor:

```
module Make (AstFilters : Camlp4.Sig.AstFilters) =
struct
 open AstFilters

 let rec filter si =
  ...

 AstFilters.register_str_item_filter begin fun si ->
  let _loc = Ast.loc_of_str_item si in
   <:str_item<
      $list: List.map filter (Ast.list_of_str_item si [])$
   >>
end

module Id =
struct
  let name = "to_of_string"
  let version = "0.1"
end ;;

let module M = Camlp4.Register.AstFilter(Id)(Make) in ()
```

Code Transformer

Code generators:

```
and to_string _loc tid cons =
 <:str_item<
   let $lid: tid ^ "_to_string"$ = function
     $list:
       List.map
         (fun c -> <:match_case< $uid: c$ -> $`str: c$ >>)
         cons$
 >>
and of_string _loc tid cons =
 <:str_item<
   let $lid: tid ^ "_of_string"$ = function
     $list:
       List.map
         (fun c -> <:match_case<
    $tup: <:patt< $`str: c$ >>$ -> $uid: c$
    (* $`str: c$ -> $uid: c$  *)
  >>)
         cons$
     | _ -> invalid_arg "bad string"
 >>
```

Compiling Code Transformer

Compiling a byte-code:

```
ocamlc -I +camlp4 -pp camlp4of -c ex5_to_of_string.ml
```

Test Program: `test3.ml`

```
type t = Foo | Bar | Baz
```

Executing transformer:

```
> camlp4o ex5_to_of_string.cmo test3.ml
type t = | Foo | Bar | Baz
let t_to_string = function | Foo -> "Foo"
  | Bar -> "Bar" | Baz -> "Baz"
let t_of_string =
  function
  | ("Foo") -> Foo
  | ("Bar") -> Bar
  | ("Baz") -> Baz
  | _ -> invalid_arg "bad string"
```

Executing Code Transformer

Executing Code Transformer with OCaml compiler:

```
ocamlc -pp 'camlp4o ex5_to_of_string.cmo' -dsource test3.ml
```

Source Output dumped by ocamlc:

```
type t =
  | Foo
  | Bar
  | Baz
let t_to_string = function | Foo  -> "Foo"
    | Bar  -> "Bar" | Baz  -> "Baz"
let t_of_string =
  function
  | "Foo" -> Foo
  | "Bar" -> Bar
  | "Baz" -> Baz
  | _ -> invalid_arg "bad string"
```

Features of Lexer

- Camlp4 lexer converts a stream of char into a stream of tokens.
- They can be built using different tools, such as `ulex` or `ocamllex`.
- Need to define `Token`, `Error`, `Filter` and `Lexer` module library.

Error Module

Used to package an exception so it can be handled generically.

```
module type Error = sig
    type t
    exception E of t
    val to_string : t -> string
    val print : Format.formatter -> t -> unit
end
```

Token Module

The type `t` denotes a abstract token that is supported by a string converter, a formatted printer and a function to determine keyword.

```
module type Token = sig
  module Loc : Loc
  type t
  val to_string : t -> string
  val print : Format.formatter -> t -> unit
  val match_keyword : string -> t -> bool
  val extract_string : t -> string
  module Filter : ... (* see below *)
  module Error : Error
end
```

The method `extract_string` would allow literal representation of token to be obtained.

Filter Module

This provides filters over token streams.

```
module Filter : sig
  type token_filter =
    (t * Loc.t) Stream.t -> (t * Loc.t) Stream.t
  type t
  val mk : (string -> bool) -> t
  val define_filter : t -> (token_filter -> token_filter) -> unit
  val filter : t -> token_filter
  val keyword_added : t -> string -> bool -> unit
  val keyword_removed : t -> string -> unit
end;
```

The `mk` method takes a predictae which is used to separately identify keywords from identifiers.

Lexer Module

This module packages up the other modules and provide the actual lexing funcion. It takes an initial location and a character stream, and returns a strean of token and location pairs.

```
module type Lexer = sig
  module Loc : Loc
  module Token : Token with module Loc = Loc
  module Error : Error
  val mk : unit ->
    (Loc.t -> char Stream.t -> (Token.t * Loc.t) Stream.t)
end
```

An Example Lexer

This lexer is for JSON where `KEYWORD` covers `true`, `false`, `null` and punctuation, while `EOI` signals end of input. We start with the internal data representation for token.

```
type token =
  | KEYWORD  of string
  | NUMBER   of string
  | STRING   of string
  | ANTIQUOT of string * string
  | EOI
end;
```

Lexing Function

Typically based on finite state machine. With `ulex`, we can use syntax extension provided to write an efficient customized lexer.

```
let rec token c = lexer
 | eof -> EOI
 | newline -> next_line c; token c c.lexbuf
 | blank+ -> token c c.lexbuf
 | '-'? ['0'-'9']+ ('.' ['0'-'9']* )?
   (('e'|'E')('+'|'-')?(['0'-'9']+))? ->
    NUMBER (L.utf8_lexeme c.lexbuf)
 | [ "{}[]:," ] | "null" | "true" | "false" ->
    KEYWORD (L.utf8_lexeme c.lexbuf)
 | '"' ->
    set_start_loc c;
    string c c.lexbuf;
    STRING (get_stored_string c)
```

Lexing Function

Continued ..

```
| "$" ->
    set_start_loc c;
    c.enc := Ulexing.Latin1;
    let aq = antiquot c lexbuf in
    c.enc := Ulexing.Utf8;
    aq
| _ -> illegal c
```

Details in jason parser github at:

`https://github.com/jaked`

Language Extension

- Camlp4 allows grammars to be extended and modified.
- Existing grammar rules can be deleted.
- New grammar rules can be inserted.
- Consider an example to allow object method chaining. Instead of:

  ```
  (obj#foo "bar")#baz
  ```

  Let us allow:

  ```
  obj#foo "bar" #baz
  ```

Deleting a Grammar Rule

Grammar rule can be deleted by specifying the symbols on the LHS of the grammar rule.

```
open Camlp4
module Id : Sig.Id =
struct
  let name = "pa_jquery"
  let version = "0.1"
end

module Make (Syntax : Sig.Camlp4Syntax) =
struct
  open Sig
  include Syntax

  DELETE_RULE Gram expr: SELF; "#"; label END;
```

Grammar Rule can be Added

Grammar rule can be added either BEFORE or AFTER an a grammar rule entry. It can be added at an existing LEVEL, either as a FIRST entry or LAST entry.

```
  EXTEND Gram
    expr: BEFORE "apply"
      [ "#" LEFTA
        [ e = SELF; "#"; lab = label ->
            <:expr< $e$ # $lab$ >> ]
      ];
  END
end

module M = Register.OCamlSyntaxExtension(Id)(Make)
```

We also need to register this specified extension.

Another Example

We often wish for a `try` construct to be evaluated for a `let` bind and to have the body of let evaluated afterwards.
Both of below does not capture the intended behavior

```
try let x = e1 in e2
with e -> h
```

```
let x =
   try e1 with e -> h
in e2
```

Lazy Evaluation of Body

A neat solution is to use a function to delay the evaluation of `let` body, as follows:

```
(try let x = e1 in (fun () -> e2)
 with e -> fun () -> h) ()
```

Let us give this solution a syntactic sugar:

```
let try x = e1 in e2
with e -> h
```

Existing Grammar Rules

Existing grammar rule in `Camlp4OCamlRevisedParser.ml` are shown below. It gives us some ideas to write extension for our new construct.

```
[ "let"; r = opt_rec; bi = binding; "in"; x = SELF ->
    <:expr< let $rec:r$ $bi$ in $x$ >>
  ...
| "try"; e = sequence; "with"; a = match_case ->
    <:expr< try $mksequence' _loc e$ with [ $a$ ] >>
```

Grammar Extension

Let us specify this syntactic extension.

```
EXTEND Gram
 expr: LEVEL "top" [
  [ "let"; "try"; r = opt_rec; bi = binding; "in";
     e = sequence; "with"; a = match_case ->
      let a =
        List.map
         (function
           | <:match_case< $p$ when $w$ -> $e$ >> ->
               <:match_case<
                   $p$ when $w$ -> fun () -> $e$
                    >>
           | mc -> mc)
          (Ast.list_of_match_case a []) in
        <:expr<
       (try let $rec:r$ $bi$ in fun () -> do { $e$ }
             with [ $list:a$ ])()
        >>
     ]
   ];
END
```

Last lecture:

- other tools for DSL
- summary of course