Extra Project Assignment - Reproducibility of Published Results

Luiza de Melo Gomes - 2021040075

1. Choose a paper presented in a recent compiler related conference, such as CGO, PLDI or CC.

I chose the paper titled “flap: A Deterministic Parser with Fused Lexing,” published at PLDI 2023. It was written by Jeremy Yallop, Ningning Xie, and Neel Krishnaswami, and it is available at: https://dl.acm.org/doi/pdf/10.1145/3591269.

2. Download the material that the authors have made publicly available for the paper. If there is no such material, but the student still wants to reproduce the paper's results, he/she can write to the authors of the paper, asking for their implementation.

Done. The materials are publicly available at https://doi.org/10.5281/zenodo.7824835.

3. Run the implementation, trying to reproduce at least one of the experiments in the paper. Even if it is not possible to reproduce the experiments, the student can still write a report about his/her tries, to claim the extra points.

Done. It was possible to reproduce some of the experiments. I will talk about this procedure and the results in the next section.

4. Write a short report about the entire procedure. This report must contain the URL of the material that has been downloaded, plus a brief description of the student's experience with it. If the student has not been able to reproduce those experiments, than the report must explain the reasons for this failure.

It is possible to reproduce 18 claims of the authors in total. Each of these are described below:

1 - The paper claims that flap has the architecture depicted in a figure:

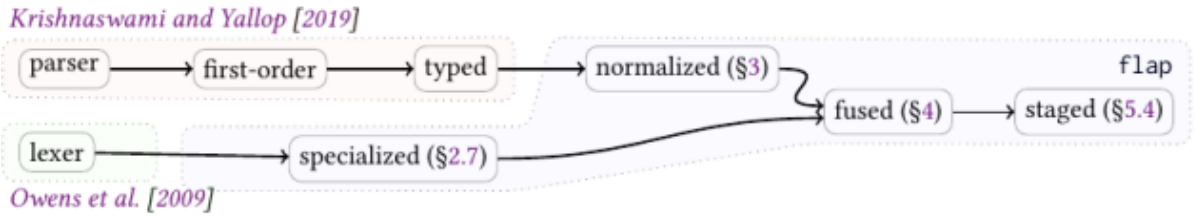

Fig. 1. Architecture of flap

We can verify this by examining the code. By running cat /home/opam/flap/lib/flap.ml we can see the first-order parser representation:

```
  type ('ctx, 'a, 'd) t' =
    Eps : 'a V.t -> ('ctx, 'a, 'd) t'
  | Seq : ('ctx, 'a, 'd) t * ('ctx, 'b, 'd) t -> ('ctx, 'a * 'b, 'd) t'
  | Tok : 'a tag -> ('ctx, 'a, 'd) t'
  | Bot : ('ctx, 'a, 'd) t'
  | Alt : ('ctx, 'a, 'd) t * ('ctx, 'a, 'd) t -> ('ctx, 'a, 'd) t'
  | Map : ('a V.t -> 'b V.t) * ('ctx, 'a, 'd) t -> ('ctx, 'b, 'd) t'
  | Fix : ('a * 'ctx, 'a, 'd) t -> ('ctx, 'a, 'd) t'
  | Var : ('ctx,'a) var -> ('ctx, 'a, 'd) t'
  | Star : ('ctx, 'a, 'd) t -> ('ctx, 'a list, 'd) t'
  and ('ctx, 'a, 'd) t = 'd * ('ctx, 'a, 'd) t'
```

It is also possible to verify the typed parser implementation in the same file:

```
  let rec typeof : type ctx a d. ctx TpEnv.t -> (ctx, a, d) t -> (ctx, a, Tp.t) t  =
    fun env (_, g) ->
    match g with
    | Eps v -> (Tp.eps, Eps v)
    | Seq (g1, g2) -> let guarded tp = {tp with Tp.guarded = true} in
                      let g1 = typeof env g1 in
                      let g2 = typeof (TpEnv.map {TpEnv.f = guarded} env) g2 in
                      (Tp.seq (data g1) (data g2), Seq(g1, g2))
    | Tok (tag,f) -> (Tp.tok tag, Tok (tag,f))
    | Bot  -> (Tp.bot, Bot)
    | Alt (g1, g2) -> let g1 = typeof env g1 in
                      let g2 = typeof env g2 in
                      (Tp.alt (data g1) (data g2), Alt(g1, g2))
    | Map(f, g') -> let g' = typeof env g' in
                    (data g', Map(f, g'))
    | Star g -> let g = typeof env g in
                (Tp.star (data g), Star g)
    | Fix g'  -> let tp = Tp.fix (fun tp -> data (typeof (tp :: env) g')) in
                 if tp.guarded then let g' = typeof (tp :: env) g' in (data g', Fix g')
                 else Format.kasprintf failwith "guarded %a" Tp.pp tp
    | Var n -> (TpEnv.lookup env n, Var n)
end
```

By running cat /home/opam/flap/lib/normal.mli it is possible to verify the normalized parser representation:

```
module Make (Term : sig type t [@@deriving ord, show] end) :
sig
  type 'a ntseq = Empty : unit ntseq
                | Cons : 'a Env.Var.t * 'b ntseq -> ('a * 'b) ntseq
  (** A possibly-empty sequence of typed variables *)

  type 'l prod = Prod : { nonterms : 'n ntseq;
                          semact : string Code.t -> 'n Code.t -> 'l Code.t; } -> 'l prod [@@deriving show]
```

By running cat /home/opam/flap/lib/flap.mli it is possible to verify the lexer interface:

```
type rhs = Skip | Error of string | Return of Term.t
val compile : (Reex.t * rhs) list -> 'a t -> ((string -> 'a) code, string) result
```

2 - The paper claims that the parser interface is the same as the one described in Krishnaswami & Yallop's 2019 article A Typed, Algebraic Approach to Parsing. I couldn't reproduce this claim, because the commands used to verify it do not show the expected output for some reason.

3 - The paper claims that the regular expressions used in flap provide various combinators, and that the lexer is constructed as a mapping from regexes to actions. I also couldn't reproduce this claim because of the same reason from claim 2.

4 - Flap's fusion produces token-free code. This can be verified by examining the code generated in the initial setting of the project:

```
match Stdlib.String.unsafe_get s_8 i_5 with
| '*'..'\255'|'!'..'\''|'\011'..'\031'|'\000'..'\b' ->
```

5 - The paper claims that flap uses MetaOCaml's staging facilities in the last step. This can be verified by examining the files code.ml and compiler.ml:

```
opam@585ee05adaf0:~/flap/lib$ cat code.ml
type 'a value = Atomic of 'a code
              | Pair : 'a value * 'b value -> ('a * 'b) value

type 'a cd = Value of 'a value
           | LetComp : 'a code * ('a code -> 'b cd) -> 'b cd

type 'a t = 'a cd

let rec fst : type a b. (a * b) cd -> a cd = function
  | Value (Atomic v) -> Value (Atomic .<Stdlib.fst .~v>.)
  | Value (Pair (x,_)) -> Value x
  | LetComp (e, k) -> LetComp (e, fun x -> fst (k x))

let rec snd : type a b. (a * b) cd -> b cd = function
  | Value (Atomic v) -> Value (Atomic .<Stdlib.snd .~v>.)
  | Value (Pair (_,y)) -> Value y
  | LetComp (e, k) -> LetComp (e, fun x -> snd (k x))

let rec pair : type a b. a cd -> b cd -> (a * b) cd =
  fun x y ->
  match x, y with
  | Value x, Value y -> Value (Pair (x, y))
  | LetComp (e, k), e2 -> LetComp (e, fun x -> pair (k x) e2)
  | Value x, LetComp (e, k) -> LetComp (e, fun y -> pair (Value x) (k y))

let inj : type a. a code -> a cd = fun e -> LetComp (e, fun x -> Value (Atomic x))
let injv : type a. a code -> a cd = fun v -> Value (Atomic v)
let rec dyn : type a. a cd -> a code = function
  | Value (Atomic c) -> c
  | Value (Pair (x, y)) -> .< (.~(dyn (Value x)), .~(dyn (Value y))) >.
  | LetComp (e, k) -> .< let x = .~e in .~(dyn (k .<x>.)) >.

let rec let_ : type a b. a cd -> (a cd -> b cd) -> b cd =
  fun e k ->
  match e with
    Value _ as v -> k v
  | LetComp (e1, k1) -> LetComp (e1, fun x -> let_ (k1 x) k)

let unit = Value (Atomic .<()>.)
opam@585ee05adaf0:~/flap/lib$
```

```
opam@585ee05adaf0:~/flap/lib$ cat code.ml
type 'a value = Atomic of 'a code
              | Pair : 'a value * 'b value -> ('a * 'b) value

type 'a cd = Value of 'a value
           | LetComp : 'a code * ('a code -> 'b cd) -> 'b cd

type 'a t = 'a cd

let rec fst : type a b. (a * b) cd -> a cd = function
  | Value (Atomic v) -> Value (Atomic .<Stdlib.fst .~v>.)
  | Value (Pair (x,_)) -> Value x
  | LetComp (e, k) -> LetComp (e, fun x -> fst (k x))

let rec snd : type a b. (a * b) cd -> b cd = function
  | Value (Atomic v) -> Value (Atomic .<Stdlib.snd .~v>.)
  | Value (Pair (_,y)) -> Value y
  | LetComp (e, k) -> LetComp (e, fun x -> snd (k x))

let rec pair : type a b. a cd -> b cd -> (a * b) cd =
  fun x y ->
  match x, y with
  | Value x, Value y -> Value (Pair (x, y))
  | LetComp (e, k), e2 -> LetComp (e, fun x -> pair (k x) e2)
  | Value x, LetComp (e, k) -> LetComp (e, fun y -> pair (Value x) (k y))

let inj : type a. a code -> a cd = fun e -> LetComp (e, fun x -> Value (Atomic x))
let injv : type a. a code -> a cd = fun v -> Value (Atomic v)
let rec dyn : type a. a cd -> a code = function
  | Value (Atomic c) -> c
  | Value (Pair (x, y)) -> .< (.~(dyn (Value x)), .~(dyn (Value y))) >.
  | LetComp (e, k) -> .< let x = .~e in .~(dyn (k .<x>.)) >.

let rec let_ : type a b. a cd -> (a cd -> b cd) -> b cd =
  fun e k ->
  match e with
    Value _ as v -> k v
  | LetComp (e1, k1) -> LetComp (e1, fun x -> let_ (k1 x) k)

let unit = Value (Atomic .<()>.)
```

It is possible to note that the files contain various occurrences of MetaOCaml's quotation (.< ... >.) and splicing (.~) constructs.

6 - The paper claims that flap uses a letrec insertion library for generating mutually-recursive functions. By examining the compiler.ml file, we can see that it contains calls to build a module Rec using letrec and calls to components of Rec:

```
module Rec = Letrec.Make(Idx)
```

```
Rec.letrec {rhs=rhs}
  (fun {resolve} ->
    .< fun s -> let start' = ref 0 and len = String.length s in
                .~(let rs, eps = lookup items start in
                   resolve (T {nt=start; prods=items; fn = eps_fallback eps;
                               nexts=List.map (fun (r, k) -> (r, unCont {resolve} items k)) rs }))
                  start' ~index:0 ~prev:0 ~len s >.)
```

7 - The paper claims that the generated code operates on OCaml's flat array representation of strings rather than on linked lists. We can verify it by examining the code generated by setting the example grammar proposed by the authors:

```
and len_3 = Stdlib.String.length s_1 in
```

```
match Stdlib.String.unsafe_get s_45 i_42 with
```

8 - The paper claims that flap optimises the end-of-input check by checking for a nullterminator rather than checking the length of the input. This can be verified using the same code of the previous claim. The code matches the input against '\000' rather than checking the length:

```
match Stdlib.String.unsafe_get s_45 i_42 with
| '*'..'\255'|'!'..'\''|'\011'..'\031'|'\000'..'\b' ->
```

9 - The paper claims that flap generates code that branches on character classes rather than on characters. This can also be verified with the code from the previous claim. We note that the functions that examine characters contain patterns such as 'u'..'\255' that match character ranges.

10 - The paper claims that OCaml compiles tail calls to known functions to efficient code. This can be verified by compiling code with tail calls and examining the generated assembly:

```
opam@585ee05adaf0:~/flap/lib$ cat /tmp/test.s
        .file ""
        .section .rodata.cst8,"a",@progbits
        .align  16
caml_negf_mask:
        .quad   0x8000000000000000
        .quad   0
        .align  16
caml_absf_mask:
        .quad   0x7fffffffffffffff
        .quad   -1
        .data
        .globl  camlTest__data_begin
camlTest__data_begin:
        .text
        .globl  camlTest__code_begin
camlTest__code_begin:
        .data
        .align  8
        .data
        .align  8
        .quad   6135
camlTest__1:
        .quad   camlTest__f_80
        .quad   3
        .quad   3321
        .quad   camlTest__g_81
        .quad   3
        .data
        .align  8
        .quad   2816
        .globl  camlTest
camlTest:
        .quad   1
        .quad   1
        .data
        .align  8
        .globl  camlTest__gc_roots
camlTest__gc_roots:
        .quad   camlTest
        .quad   0
        .text
        .align  16
        .globl  camlTest__f_80
camlTest__f_80:
        .cfi_startproc
.L103:
        movq    %rax, %rbx
        cmpq    $1, %rbx
        je      .L102
        movl    $4, %eax
        subq    %rbx, %rax
        jmp     camlTest__g_81@PLT
        .align  4
.L102:
```

I won’t show all the code, but it is possible to verify that the assembly code do not contain call instructions.

11 - The paper claims that the evaluation compares six parser implementations:

(a) ocamlyacc

(b) menhir in table-generation mode

(c) menhir in code-generation mode

(d) flap

(e) asp [Krishnaswami and Yallop 2019]

(f) ParTS [Casinghino and Roux 2020]

This can be verified by examining the subdirectories of /home/opam/flap/benchmarks:

```
opam@585ee05adaf0:~/flap/benchmarks$ ls
README.md  common  csv  intexp  json  munge.py  pgn  ppm  sexp
```

```
opam@585ee05adaf0:~/flap/benchmarks/json$ ls
README.md  dune            json_lexer.mll           json_lexer_menhir_table.mll  json_parser_menhir_code.mly   json_parts.ml                 json_tokens.ml   json_tokens_base.ml
data       json_benchmark.ml  json_lexer_menhir_code.mll  json_parser.mly           json_parser_menhir_table.mly  json_staged_combinator_parser.ml  json_tokens.mli  json_unstaged_combinator_parser.ml
opam@585ee05adaf0:~/flap/benchmarks/json$
```

12 - The authors claim that flap has the best throughput of the implementations evaluated. This can be verified by running the benchmarks with `make bench`:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

28 normalized nonterminals
55 normalized productions
83 fused productions
[intexp]: compilation time : 631.519ms

  Name                          Time R^2   Time/Run              95ci
 ----------------------------- ---------- ---------- -----------------
  ocamlyacc_intexp:262144           0.99    11.88ms    -0.09ms +0.12ms
  ocamlyacc_intexp:524288           0.97    26.13ms    -1.03ms +0.89ms
  ocamlyacc_intexp:786432           0.99    39.04ms    -0.57ms +0.86ms
  ocamlyacc_intexp:1048576          0.99    53.18ms    -1.13ms +1.55ms
  ocamlyacc_intexp:1310720          0.99    66.30ms    -1.38ms +1.80ms
  ocamlyacc_intexp:1572864          0.99    75.60ms    -2.26ms +2.70ms
  ocamlyacc_intexp:1835008          1.00    85.41ms    -1.16ms +1.55ms
  ocamlyacc_intexp:2097152          1.00    99.13ms    -1.56ms +1.73ms
  menhir_code_intexp:262144         0.98     7.02ms    -0.12ms +0.14ms
  menhir_code_intexp:524288         0.99    14.68ms    -0.31ms +0.39ms
  menhir_code_intexp:786432         1.00    21.56ms    -0.15ms +0.15ms
  menhir_code_intexp:1048576        0.86    34.77ms    -3.13ms +4.51ms
  menhir_code_intexp:1310720        0.89    47.10ms    -3.49ms +4.13ms
  menhir_code_intexp:1572864        0.95    53.77ms    -3.21ms +3.43ms
  menhir_code_intexp:1835008        0.84    72.81ms   -11.53ms +9.73ms
  menhir_code_intexp:2097152        0.95    93.65ms    -4.75ms +5.13ms
  menhir_table_intexp:262144        0.90    19.38ms    -0.80ms +0.94ms
  menhir_table_intexp:524288        0.93    39.67ms    -2.36ms +2.48ms
  menhir_table_intexp:786432        0.87    77.49ms   -7.28ms +10.73ms
  menhir_table_intexp:1048576       0.99    86.20ms    -1.93ms +2.18ms
  menhir_table_intexp:1310720       0.95    89.73ms    -4.27ms +5.89ms
  menhir_table_intexp:1572864       1.00   118.19ms    -1.94ms +1.77ms
  menhir_table_intexp:1835008       1.00   123.90ms    -2.20ms +2.41ms
  menhir_table_intexp:2097152       1.00   130.97ms    -1.80ms +3.06ms
  staged_intexp:262144              1.00    11.63ms    -0.07ms +0.07ms
  staged_intexp:524288              1.00    23.41ms    -0.16ms +0.16ms
  staged_intexp:786432              1.00    35.93ms    -0.25ms +0.28ms
  staged_intexp:1048576             1.00    47.09ms    -0.26ms +0.26ms
  staged_intexp:1310720             1.00    59.26ms    -0.34ms +0.36ms
  staged_intexp:1572864             1.00    70.36ms    -0.34ms +0.37ms
  staged_intexp:1835008             1.00    82.23ms    -0.34ms +0.37ms
  staged_intexp:2097152             1.00    95.81ms    -1.06ms +0.84ms
  fused_intexp:262144               1.00     4.66ms    -0.04ms +0.04ms
  fused_intexp:524288               1.00    10.39ms    -0.07ms +0.07ms
  fused_intexp:786432               1.00    16.37ms    -0.16ms +0.17ms
  fused_intexp:1048576              1.00    21.08ms    -0.20ms +0.24ms
  fused_intexp:1310720              0.99    27.54ms    -0.45ms +0.58ms
  fused_intexp:1572864              1.00    33.12ms    -0.22ms +0.26ms
  fused_intexp:1835008              1.00    39.76ms    -0.39ms +0.46ms
  fused_intexp:2097152              1.00    42.66ms    -0.40ms +0.43ms

Benchmarks that take 1ns to 100ms can be estimated precisely. For more reliable
estimates, redesign your benchmark to have a shorter execution time.
```

For each benchmark, for each input size, the fused implementation have the lowest running time.

The authors also provide a script that calculates the throughputs from the time recorded. By running python throughput.py it is possible to verify that the fused implementation has the highest throughput for most of the benchmarks:

```
opam@72a7e95dae2b:~/flap$ python throughput.py
benchmark,fused,staged,ocamlyacc,menhir_table,menhir_code,parts,menhir_normalized_table,menhir_normalized_code,ocamlyacc_normalized,normalized
json,936.3192619213645,114.44925255704169,192.58809770736335,150.56020838544404,265.5722740342923,31.911416286416287,0,0,0,0
sexp,170.63889340927585,73.455411558669,58.6287950796757,44.916513171985436,89.27850148999575,43.536474984430136,0,0,0,0
arith,49.147327707454295,21.88315415927356,21.150257237970344,16.008437046651903,22.387880405766147,0,0,0,0,0
pgn,159.54913336801886,46.36997059398581,44.57527476477699,33.11402630766955,74.37261948454626,0,0,0,0,0
ppm,73.46278671687779,18.14484024306676,12.350096275416757,11.251604751890309,32.90192874771234,0,0,0,0,0
csv,205.62755245589952,0,43.72431284887841,40.39999899493531,95.66760081792776,0,0,0,0,0
opam@72a7e95dae2b:~/flap$
```

13 - The paper claims that the benchmark implementations use either ocamllex or combinators. This can be verified by examining the benchmark code:

```
let lex =
  let open Json_tokens_base in
  fix @@ fun lex ->
    (chr '[' $ (fun _ -> .< Some (T (LBRACKET, ())) >.))
<|> (chr ']' $ (fun _ -> .< Some (T (RBRACKET, ())) >.))
<|> (chr '{' $ (fun _ -> .< Some (T (LBRACE, ())) >.))
<|> (chr '}' $ (fun _ -> .< Some (T (RBRACE, ())) >.))
<|> (chr ',' $ (fun _ -> .< Some (T (COMMA, ())) >.))
<|> (chr ':' $ (fun _ -> .< Some (T (COLON, ())) >.))
<|> (chr 'n' >>>
     chr 'u' >>>
     chr 'l' >>>
     chr 'l' $ fun _ -> .< Some (T (NULL, ()))>.)
<|> (chr 't' >>>
     chr 'r' >>>
     chr 'u' >>>
     chr 'e' $ fun _ -> .<Some (T (TRUE, ()))>.)
<|> (chr 'f' >>>
     chr 'a' >>>
     chr 'l' >>>
     chr 's' >>>
     chr 'e' $ fun _ -> .<Some (T (FALSE, ()))>.)
<|> (string $ (fun s -> .<Some (T (STRING, .~s))>.))
<|> (decimal $ (fun s -> .<Some (T (DECIMAL, .~s))>.))
<|> (charset " \t\r\n" >>>
     lex $ fun p -> .< snd .~p >.)
<|> eps .<None>.
```

14 - The paper claims that the evaluation uses identically-structured parsers for ocamlyacc and menhir and identically-structured parsers for ParTS, asp and flap. This can also be verified by examining the benchmark code. For the json benchmark for example, we can confirm that the parsers for the first three benchmarks are identical, since two of the files are symbolic links to the other:

```
opam@585ee05adaf0:~/flap$ ls -l benchmarks/json/*parser*mly
-rw-r--r-- 1 root root 1329 Apr 12  2023 benchmarks/json/json_parser.mly
lrwxrwxrwx 1 root root   15 Apr 12  2023 benchmarks/json/json_parser_menhir_code.mly -> json_parser.mly
lrwxrwxrwx 1 root root   15 Apr 12  2023 benchmarks/json/json_parser_menhir_table.mly -> json_parser.mly
opam@585ee05adaf0:~/flap$
```

15 - The paper claims that flap and the other implementations have linear-time parsing. This can be verified by examining the ratios between input sizes and running times in the figures reported in claim 12:

```
fused_intexp:262144              1.00     4.66ms    -0.04ms +0.04ms
fused_intexp:524288              1.00    10.39ms    -0.07ms +0.07ms
fused_intexp:786432              1.00    16.37ms    -0.16ms +0.17ms
fused_intexp:1048576             1.00    21.08ms    -0.20ms +0.24ms
fused_intexp:1310720             0.99    27.54ms    -0.45ms +0.58ms
fused_intexp:1572864             1.00    33.12ms    -0.22ms +0.26ms
fused_intexp:1835008             1.00    39.76ms    -0.39ms +0.46ms
fused_intexp:2097152             1.00    42.66ms    -0.40ms +0.43ms
```

16 - The paper claims that the evaluation is based on six benchmarks. This can be verified by examining the six subdirectories of the /home/opam/flap/benchmarks directory, which correspond to the six benchmarks described in the paper:

```
opam@585ee05adaf0:~/flap$ ls /home/opam/flap/benchmarks
README.md  common  csv  intexp  json  munge.py  pgn  ppm  sexp
```

17 - The paper claims that normalized grammars have certain sizes. This can be verified by running the example grammars provided by the authors. For example, the results of table 1 of the paper for the sexp grammar, it's possible to verify the numbers by running the example grammars:

```
# Sexp_grammar.code;;
3 normalized nonterminals
6 normalized productions
9 fused productions
[sexp]: compilation time : 1.718ms
- : unit = ()
#
```

18 - The authors claim that compilation time is acceptable. We can verify this by running dune runtest -f:

```
opam@585ee05adaf0:~/flap$ dune runtest -f
4 lexing rules
ee expressions
nonterminals
6 normalized productions
9 fused productions
[sexp]: compilation time : 0.401ms
13 lexing rules
95 context-free expressions
38 normalized nonterminals
53 normalized productions
91 fused productions
[pgn]: compilation time : 324.96ms
6 lexing rules
10 context-free expressions
5 normalized nonterminals
6 normalized productions
16 fused productions
[ppm]: compilation time : 5.606ms
12 lexing rules
42 context-free expressions
9 normalized nonterminals
33 normalized productions
42 fused productions
[json]: compilation time : 44.311ms
3 lexing rules
14 context-free expressions
5 normalized nonterminals
7 normalized productions
7 fused productions
[csv]: compilation time : 0.99ms
14 lexing rules
143 context-free expressions
28 normalized nonterminals
55 normalized productions
83 fused productions
[intexp]: compilation time : 718.882ms
4 lexing rules
11 context-free expressions
File "/tmp/build_ce2c87_dune/runnac41ce.ml", line 37, characters 44-50:
37 |                                           let x_5187 =
                                                  ^^^^^^

Warning 26: unused variable x_5187.
File "/tmp/build_ce2c87_dune/runnac41ce.ml", line 76, characters 44-50:
76 |                                           let x_5184 =
                                                  ^^^^^^

Warning 26: unused variable x_5184.
3 normalized nonterminals
6 normalized productions
9 fused productions
3 lexing rules
```

7824835/flap.tgz

```
Warning 26: unused variable x_5184.
3 normalized nonterminals
6 normalized productions
9 fused productions
3 lexing rules
8 context-free expressions
....1 normalized nonterminals
3 normalized productions
3 fused productions
3 normalized nonterminals
6 normalized productions
9 fused productions
1 normalized nonterminals
3 normalized productions
3 fused productions
1 normalized nonterminals
3 normalized productions
3 fused productions
3 normalized nonterminals
6 normalized productions
9 fused productions
1 normalized nonterminals
3 normalized productions
3 fused productions
1 normalized nonterminals
3 normalized productions
3 fused productions
3 normalized nonterminals
6 normalized productions
9 fused productions
1 normalized nonterminals
3 normalized productions
3 fused productions
1 normalized nonterminals
3 normalized productions
3 fused productions
3 normalized nonterminals
6 normalized productions
9 fused productions
1 normalized nonterminals
3 normalized productions
3 fused productions

Ran: 4 tests in: 0.13 seconds.
OK
1 normalized nonterminals
3 normalized productions
3 fused productions
3 normalized nonterminals
6 normalized productions
9 fused productions
1 normalized nonterminals
3 normalized productions
3 fused productions
opam@585ee05adaf0:~/flap$
```

In the output it is possible to note that all tests runned in 0.13 seconds, which is an acceptable compilation time.

Overall, I had a very positive experience with reproducing the experiments of the paper. The instructions of the authors to reproduce the experiments were very clear and easy to follow.