Step-by-step instructions

`flap`

A deterministic parser with fused lexing

Artifact evaluation

The `flap` authors

11th April 2023

Introduction

This document is a guide to using the artifact that accompanies the PLDI 2023 paper *`flap`: a deterministic parser with fused lexing*. It has three sections besides this introduction:

Tutorial example (pages 3–5) gives a small example of a fused grammar implemented in `flap`, which the reader can execute at the MetaOCaml top-level.

It assumes that the reader has already worked through the Getting Started Guide.

Claims supported by the artifact (pages 6–22) presents claims supported by the artifact accompanying this document.

We have identified eighteen claims related to the artifact in the submitted paper. The section shows how to locate and examine the evidence that supports each claim.

For each claim the section includes the excerpt from the paper where the claim is made, together with line numbers that make the excerpt easy to locate.

Each claim also comes with an indication of the length of time we estimate it will take to verify it. We estimate that verifying all the claims will take approximately three hours

The section assumes that the reader has already worked through the Tutorial example section.

Claims not supported by the artifact (page 23) lists claims in the paper that are not supported by the artifact, but are instead supported by proofs in supplementary material.

Tutorial example

This section guides the reader through an interactive exploration of the library via an example, constructing a simple fused lexer and parser for the Dyck language of balanced brackets.

1. Start the MetaOCaml toplevel

   ```
   metaocaml
   ```

   `flap` is implemented a MetaOCaml library, and all its functionality is available from the top-level read-eval-print loop.

2. Within MetaOCaml load the `topfind` command, then load `flap`:

   ```
   #use "topfind";;
   #require "flap";;
   ```

   The `topfind` command is used to locate and load libraries into the top-level.

   The final line of the output should indicate that `flap` has been loaded:

   ```
   /home/opam/.opam/4.11.1+BER/lib/flap/flap.cma: loaded
   ```

3. Load the `ppx_deriving` extension that automatically creates pretty-printers and comparison functions:

   ```
   #require "ppx_deriving.std";;
   ```

   MetaOCaml's response should end as follows:

   ```
   ppx_deriving: package:ppx_deriving.std: option added
   ```

4. Create a type of tokens and instantiate the `flap` parser with the type:

   ```
   module Tok = struct type t = LPAREN | RPAREN [@@deriving ord, show] end;;
   module P = Flap.Parse(Tok);;
   ```

   (From this point on, MetaOCaml should respond by printing out the types and values of the variables defined.)

   As in many parsing systems, these tokens will serve as an interface between the lexer and the parser. The distinctive feature of `flap` is that the tokens are only used during compilation, and are not present in the final parser.

5. Define a lexer as an ordered mapping from regexes to actions (see lines 168–170 of our paper):

```
let lexer = [
    Reex.chr '('          , P.Return LPAREN;
    Reex.chr ')'          , P.Return RPAREN;
    Reex.regex "[\t\n ]" , P.Skip;
  ];;
```

   The semantics of lexers are similar to standard lexers, such as `lex`: the lexer finds the longest match, with earlier patterns taking priority. Here there are two types of action: `Return` (pass a token to the parser) and `Skip` (discard the matched portion of the text and restart lexing at the text that follows).

6. Add code that builds payloads for each token.

```
let ignore _ = Flap.Cd.injv .<()>.
let lparen = P.tok LPAREN @@ ignore
let rparen = P.tok RPAREN @@ ignore
;;
```

   This simple example always returns empty payloads (i.e. `()`). Larger and more realistic lexers written with `flap` might instead return values (such as integers or strings) constructed from the matched text.

7. Define the `star` combinator:

```
let star_ e =
  let open Flap.Cd in
  let open P in fix @@ fun x -> (eps (injv .<[]>.)
            <|> (e >>> x $ fun p -> let_ p @@ fun p ->
                               injv .< .~(dyn (fst p)) :: .~(dyn (snd p)) >.))
;;
```

   This is a temporary measure: `flap` currently provides a set of basic parsing combinators, which can be used to construct more complex combinators such as Kleene `star` and `plus`. Before releasing `flap` we plan to incorporate a selection of these definitions into the library to make things more convenient for users.

   Here `star e` is built as a fixpoint $\mu x.\epsilon \mid e \cdot x$; however, it is not necessary to understand the details in order to evaluate the artifact.

8. Define the parser using `flap`'s combinators

```
let parser =
  let open P in
  let open Flap.Cd in
  fix @@ fun vd ->
   ((lparen >>> star_ vd >>> rparen)
    $ fun p -> injv .< 1 + List.fold_left max 0
                           (Stdlib.snd (Stdlib.fst .~(dyn p))) >.)
;;
```

The parser is again built using a fixpoint: $\mu vd.(\cdot vd * \cdot)$.

The semantic action of the parser measures the "depth" of the Dyck sentence, defined as the successor of the maximum depth of its children.

9. Compile the parser and lexer and extract the code from the result:

```
let result = P.compile lexer parser;;
let Ok code = result;;
```

MetaOCaml should print the full code for the generated parser.

(There will also be a warning, "pattern-matching is not exhaustive", because the second line of the code only matches the `Ok` constructor, not the `Error` constructor that can be returned if `P.compile` fails. The warning can be safely ignored.)

```
  ...
  let rec x_1 start_4 ~index:i_5  ~prev:prev_6  ~len:len_7  s_8 =
    match Stdlib.String.unsafe_get s_8 i_5 with
    | 'u'..'\255'|'o'..'s'|']'..'m'|'*'..'['|'!'..'\''|'\000'..'\031' ->
```

The `P.compile` function type-checks the parser (§2.1 of the paper), normalizes the result (§2.6), and fuses the lexer and parser together, before finally generating code using MetaOCaml and letrec insertion.

If you inspect the generated code, you will not find any trace of the token type, because it is eliminated by fusion. The claims enumerated in the next section gives more detail; for example, Claim 4 investigates token elimination.

10. Save the generated code to a file and load it into the top level:

```
let fd = open_out "/tmp/dyck.ml";;
let fmt = Format.formatter_of_out_channel fd;;
Format.fprintf fmt "let dyck = %a@."
   Codelib.format_code (Codelib.close_code code);;
#mod_use "/tmp/dyck.ml";;
```

11. Try out the generated parser:

```
Dyck.dyck "(()(()))";;
```

The response should be 3, the maximum depth of the Dyck sentence.

Claims supported by the artifact

§1 Introduction

Claim 1: Architecture of `flap` **(10 minutes)**

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

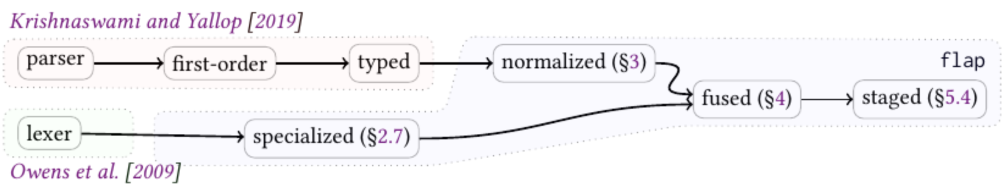

Fig. 1. Architecture of flap

You can verify this claim by examining the code. The components of `flap` are found in the following locations, all below the `/home/opam/flap/lib` directory:

The parser interface is found in `flap.mli`, which includes combinators `eps`, `star`, etc., which are described in detail in the paper.

The first-order parser representation is found in `flap.ml`:

From `/home/opam/flap/lib/flap.ml`

```
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'
```

The typed parser representation is also found in `flap.ml`: it is a variant of the first-order representation constructed by the `typeof` function:

From `/home/opam/flap/lib/flap.ml`

```
let rec typeof : type ctx a d. ctx TpEnv.t -> (ctx, a, d) t -> (ctx, a, Tp.t) t  =
```

The normalized parser representation is found in `normal.mli`:

From /home/opam/flap/lib/normal.mli

```
  This corresponds to the normal form defined in Figure 4 of

      flap: A Deterministic Parser with Fused Lexing
      PLDI 2023
 *)
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;
```

The lexer interface is found in `flap.mli`. A lexer is a list of pairs of regular expressions and actions, discussed in more detail later in this document:

From /home/opam/flap/lib/flap.mli

```
  type rhs = Skip | Error of string | Return of Term.t
  val compile : (Reex.t * rhs) list -> 'a t -> ((string -> 'a) code, string) result
  (** [compile l p] builds code [Ok c] for a lexer [l] and type-checked parser [p],
```

The `fuse` function in `fused.mli` carries out lexer specialization and fusion:

From /home/opam/flap/lib/fused.mli

```
  (** A lexer right-hand side is an action: either [Skip] (restart lexing),
```

The staged representation is generated by `compiler.ml`. Its implementation in terms of staging features and letrec insertion is covered in a little more detail later in this document.

§2 Overview

Claim 2: The parser interface **(1–2 minutes)**

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*:

> Since every well-typed context free expression normalizes to DGNF, we can provide the same parser combinator interface as Krishnaswami and Yallop, but with a significantly more efficient implementation (§6).

§2.1 of A Typed, Algebraic Approach to Parsing presents the seven fundamental combinators of the parsing interface: `eps`, `chr`, `seq`, `bot`, `alt`, `fix`, `map`.

You can see the implementation of these combinators in the asp library that accompanies that paper by executing the following commands in the `metaocaml` toplevel:

```
#use "topfind";;
#require "asp";;
#show Asp.Staged.Parse;;
```

The output should include the following:

```
...
      type 'a t
      val eps : 'a code -> 'a t
      val ( >>> ) : 'a t -> 'b t -> ('a * 'b) t
      val tok : 'a Token.tag -> 'a t
      val bot : 'a t
      val ( <|> ) : 'a t -> 'a t -> 'a t
      val any : 'a t list -> 'a t
      val ( $ ) : 'a t -> ('a code -> 'b code) -> 'b t
      val fix : ('b t -> 'b t) -> 'b t
...
```

There are some differences between their paper and their implementation in the names and in the types of combinators. In the implementation `seq` is called >>>, `alt` is called <|>, `chr` is called `tok`, `map` is called $.

The `flap` implementation provides combinators that use the same names as `asp` and that have corresponding types, as you can confirm by executing the following commands in the `metaocaml` toplevel:

```
#use "topfind";;
#require "flap";;
#show Flap.Parse;;
```

The output should include the following

```
...
      type _ t
      val eps : 'a Flap.Cd.t -> 'a t
      val ( >>> ) : 'a t -> 'b t -> ('a * 'b) t
      val tok : Term.t -> (string Flap.Cd.t -> 'a Flap.Cd.t) -> 'a t
      val bot : 'a t
      val ( <|> ) : 'a t -> 'a t -> 'a t
      ...
      val ( $ ) : 'a t -> ('a Flap.Cd.t -> 'b Flap.Cd.t) -> 'b t
      val fix : ('b t -> 'b t) -> 'b t
...
```

You can also examine the combinators and their documentation in `/home/opam/flap/lib/flap.mli`:

From `/home/opam/flap/lib/flap.mli`

```
type _ t
(** The type of parsers *)

val eps : 'a Code.t -> 'a t
(** [eps v] succeeds without consuming input and returns [v] *)

val ( >>> ) : 'a t -> 'b t -> ('a * 'b) t
(** [p >>> q] parses successive prefixes of the input using [p] and then [q]
    and returns a pair of the result of the two parses *)
```

Claim 3: The lexer interface (5 minutes)

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:

> *Lexer.* We start with the lexer. Fig. 3a defines the syntax for regexes r and lexers L. Regexes r include $\bot$ for nothing , ϵ for the empty string, characters c, sequencing $r \cdot s$, alternation $r \mid s$, Kleene star $r*$, intersection $r \mathbin{\&} s$, and negation $\neg r$. A lexer L is an ordered mapping from regexes to *actions*, where an action might return a token ($r \Rightarrow$ **Return** t), invoke the lexer recursively to skip over some input $r \Rightarrow$ **Skip**, or raise an error otherwise. Our example sexp lexer (Fig. 3b) has four actions: three return tokens ATOM, LPAR and RPAR, and one skips whitespace.

You can verify the part of the claim about regexes by loading our `reex` library into the toplevel:

```
#use "topfind";;
#require "reex";;
#show Reex;;
```

The output should include the following lines:

```
val empty : t
val epsilon : t
...
val ( <&> ) : t -> t -> t
...
val ( >>> ) : t -> t -> t
...
val ( <|> ) : t -> t -> t
...
val star : t -> t
val not : t -> t
...
val chr : char -> t
```

These combinators correspond to the regex forms described in the paper:

$\bot$	`empty`
ϵ	`epsilon`
c	`chr`
$r \cdot s$	`>>>`
$r \mid s$	`<|>`
$r*$	`star`
$r \;\&\; s$	`<&>`
$\neg r$	`not`

The other parts of the claim involve lexer actions. The three types of lexer actions appear in the `rhs` type in the interface in `flap.mli`:

From `/home/opam/flap/lib/flap.mli`

```
type rhs = Skip | Error of string | Return of Term.t
```

The first argument of the `compile` function in `flap.mli` is a list of pairs of regular expressions and actions (type `rhs`) that corresponds to the mapping described in the paper:

From `/home/opam/flap/lib/flap.mli`

```
val compile : (Reex.t * rhs) list -> 'a t -> ((string -> 'a) code, string) result
```

The Dyck parser developed on page 3 of this document gives a concrete example of a `flap` lexer.

Claim 4: `flap`'s fusion produces token-free code (10 minutes)

A central claim of the paper is that fusion produces token-free code:

> Fusion acts on a lexer and a normalized parser, connected via tokens, and produces a grammar that is entirely token-free, in which the only branches involve inspecting individual characters.

You can verify this claim by examining the code generated by the example grammar on pages 3–5 of this document. The code contains three branches (all instances of `match`), all of which operate on characters, e.g.:

```
match Stdlib.String.unsafe_get s_8 i_5 with
| 'u'..'\255'|'o'..'s'|']'..'m'|'*'..'['|'!'..'\''|'\000'..'\031' ->
```

The fact that fusion operates on a separately-defined lexer and normalized parser can be seen from the type of the `fuse` function:

From `/home/opam/flap/lib/fused.mli`

```
(** A lexer right-hand side is an action: either [Skip] (restart lexing),
```

The first argument to `fuse` is a lexer (defined as a list of pairs of regular expressions and actions). The second argument is a normalized grammar. The result is a fused grammar.

Claim 5: The last step: staging (1–2 minutes)

The paper claims that `flap` uses MetaOCaml's staging facilities in the last step:

> In the last step, `flap` uses MetaOCaml's staging facilities to generate code for the fused grammar.

You can verify this claim by examining the files `code.ml` and `compiler.ml` in the directory `/home/opam/flap/lib`.

The files contain various occurrences of MetaOCaml's quotation (`.< ... >.`) and splicing (`.~`) constructs.

You might also like to check that quotation and splicing are not used in earlier stages (type checking, normalization, fusion, etc.) by examining the other files in that directory.

§5 Implementation of Parsing

Claim 6: `flap` uses `letrec` (2 minutes)

The paper claims that `flap` uses a *letrec insertion* library for generating mutually-recursive functions:

> `flap` generates code for fused grammars using MetaOCaml's staging facilities together with Yallop and Kiselyov's [2019] *letrec insertion* library for creating the indexed mutually-recursive functions produced by the staged parsing algorithm (§5.4).

You can verify this claim by examining the file `/home/opam/flap/lib/compiler.ml`, which contains calls to build a module `Rec` using the `Letrec` module from the *letrec insertion* library

From `/home/opam/flap/lib/compiler.ml`

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

and calls to the various components of `Rec`, e.g.:

From `/home/opam/flap/lib/compiler.ml`

```
Rec.letrec {rhs=rhs}
  (fun {resolve} ->
```

The next three claims involve examining the output of code generated by `flap`.

Claim 7: `flap` operates on flat arrays (5 minutes)

The paper claims that the generated code operates on OCaml's flat array representation of strings rather than on linked lists:

Second, while the input to the pseudocode is a character linked list, `flap` operates on OCaml's flat array representation of strings, using indexes to keep track of string positions as parsing proceeds.

You can verify this claim by examining the code generated by the example grammar on pages 3–5 of this document. The code contains calls to the functions `String.unsafe_get` and `String.length`, which operate on OCaml's standard flat array representation of strings:

```
match Stdlib.String.unsafe_get s_8 i_5 with
...
and len_3 = Stdlib.String.length s_1 in
```

Claim 8: `flap` optimises the end-of-input test (5 minutes)

The paper claims that `flap` optimises the end-of-input check by checking for a null-terminator rather than checking the length of the input:

Relatedly, `flap` also optimizes the end of input test by using the fact that OCaml's strings are null-terminated, like C's. This representation allows the end of input check to be incorporated into the per-character branch in the generated code: a null character `'\000'` indicates a *possible* end of input, which can subsequently be confirmed by checking the string length.

You can verify this claim by examining the code generated by the example grammar on pages 3–5 of this document. The generated code matches the input against the null character `'\000'` rather than checking the length:

```
match Stdlib.String.unsafe_get s_45 i_42 with
| 'u'..'\255'|'o'..'s'|']'..'m'|'*'..'['|'!'..'\''|'\000'..'\031' ->
```

The example grammar specifies the behaviour on end-of-input to be the same as the behaviour for the null character, so it is unnecessary to confirm the length after encountering null, and the generated code does not do so. In other cases, confirming the length is necessary, and our library `reex`, used in `flap`, will generate code to do so. For example, the following code creates a matcher for the language that accepts only the null character:

```
#use "topfind";;
#require "reex_match";;
Reex_match.match_ ~options:{Reex_match.default_options with match_type = `ranges}
   .<0>. .< "">.
   [Reex.chr '\000', fun _ ~index ~len _ -> .< 1 >.]
;;
```

In the case where the input matches `'\000'`, the generated code subsequently checks the string length to decide whether to reject (if end-of-input has been encountered) or to accept:

```
| '\000' ->
    if i_3 = len_5
    then Stdlib.failwith "no match"
    else ...
```

Claim 9: `flap` groups character patterns into classes (5 minutes)

The paper claims that `flap` generates code that branches on character classes rather than on characters:

> Third, while the pseudocode generates a case in each branch for each possible character in the input, `flap` generates a smaller number of cases by grouping characters with equivalent behaviour into classes, as described in detail by Owens et al. [2009]. Branching on these character classes rather than treating characters individually leads to a substantial reduction in code size.
>
> Here is an excerpt of the code generated by `flap` for the s-expression parser:

```
and parse5 r i len s = match s.[i] with
   | ' '|'\n' → parse6 r (i + 1) len s
   | '('      → parse9 r (i + 1) len s
   | 'a'..'z' → parse3 r (i + 1) len s
   | '\000'   → if i = len then [] else failwith "unexpected"
   | _        → []
```

You can verify this claim by examining the code generated by the example grammar on pages 3–5 of this document. The generated functions that examine characters contain patterns such as `'u'..'\255'` that match character ranges:

```
match Stdlib.String.unsafe_get s_8 i_5 with
| 'u'..'\255'|'o'..'s'|']'..'m'|'*'..'['|'!'..'\''|'\000'..'\031' ->
```

Claim 10: OCaml compiles certain tail calls to jumps (5 minutes)

The paper claims that OCaml compiles tail calls to known functions to efficient code:

> OCaml compiles tail calls to known functions such as $\mathtt{parse}_6$ to unconditional jumps. As §6

You can verify this claim by compiling some code with tail calls and examining the generated assembly code. For example, running the following at the bash prompt will generate a file `/tmp/test.s` with code for `f` and `g`:

```
echo 'let rec f y = if y then g (not y) else g y and g y = y && f y' > /tmp/test.ml
ocamlopt -c  -S /tmp/test.ml
```

The generated code `/tmp/test.s` should contain no `call` instructions, but should contain `jmp` instructions, such as the following code that corresponds to a call to `g` from the function `f`:

```
jmp camlTest__g_81@PLT
```

You might also like to confirm that OCaml generates similarly efficient code for the tail-recursive functions generated by `flap`.

§6 Evaluation

§6 of our paper describes our quantitative evaluation of `flap`. This section shows how to verify our claims about the implementation and results of our evaluation.

Claim 11: Our evaluation is based on six implementations (5 minutes)

The paper claims that the evaluation compares six parser implementations:

> *Benchmarks.* We compare six implementations. All six guarantee deterministic, linear-time parsing, and use staging, generating code specialized to a given grammar. Our aim is to evaluate whether `flap` is faster than other asymptotically-efficient systems, so it is not possible to make meaningful comparisons with systems with different complexity (e.g. GLR or backtracking recursive-descent):
>
> (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]

You can verify this claim by examining the files in the subdirectories of `/home/opam/flap/benchmarks`. For example, the `/home/opam/flap/benchmarks/json` contains the following files:

```
json_lexer.mll                          # OCamlyacc lexer
json_parser.mly                         # OCamlyacc parser
json_lexer_menhir_code.mll              # Menhir lexer (code)
json_parser_menhir_code.mly             # Menhir parser (code)
json_lexer_menhir_table.mll             # Menhir lexer (table)
json_parser_menhir_table.mly            # Menhir parser (table)
json_parts.ml                           # ParTS lexer+parser
json_staged_combinator_parser.ml        # Asp lexer+parser
```

The implementation of the `flap` `json` parser is in a different directory: `/home/opam/flap/grammars/json_gramma`

The file `/home/opam/flap/benchmarks/json` contains code to benchmark the various implementations:

From /home/opam/flap/benchmarks/json/json_benchmark.ml

```
    | x :: (x' :: _ as xs) -> x = x' && alleq xs in
  let run n (_name, p) = Core.Staged.unstage (p n) () in
  let parsers = ["yacc"        , ocamlyacc_json;
                 "normalized_yacc"        , ocamlyacc_normalized_json;
                 "parts"        , parts_json;
                 "menhir_code" , menhir_code_json;
                 "menhir_table", menhir_table_json;
                 "menhir_normalized_code" , menhir_normalized_code_json;
                 "menhir_normalized_table", menhir_normalized_table_json;
                 "staged"       , staged_json;
                 "unstaged"     , unstaged_json;
                 "fused"        , fused_json;
                 "normalized"   , normalized_json] in
  List.iter (fun n -> assert (alleq (List.map (run n) parsers))) args;
  Gc.compact ()

open Core
open Core_bench


let () =
  Command.run (Bench.make_command [
```

The code for the other benchmarks is structured similarly.

Claim 12: `flap` has the best throughput (20–30 minutes)

The paper claims that `flap` has the best throughput of the implementations evaluated:

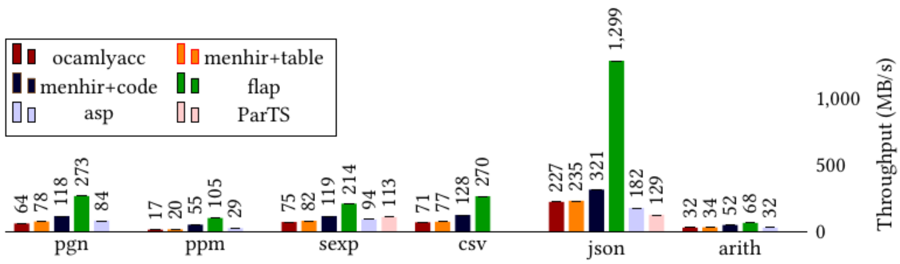

Fig. 11. Parser throughput: ocamlyacc, menhir, `flap`, asp and ParTS

As Fig. 11 shows, our experiments confirm the results reported by Krishnaswami and Yallop: the staged implementation of typed CFEs in asp generally outperforms `ocamlyacc`. The addition of lexer-parser fusion makes `flap` considerably faster than both asp and `ocamlyacc`, reaching around 1.3GB/s (a little under 2.5 cycles per byte) on the `json` benchmark. The throughput ratios of `flap` to asp ($\frac{273}{84}$ = 3.3×, $\frac{105}{29}$ = 3.6×, $\frac{124}{94}$ = 1.3×, $\frac{1299}{182}$ = 7.1×, $\frac{68}{32}$ = 2.1×) indicate the additional performance benefit provided by the combination of fusion and staging over staging alone.

You can verify this claim by running the benchmarks. Type

```
make bench
```

in the `/home/opam/flap` directory. It is safe to ignore the `unused variable` warnings.

Each benchmark will take a few minutes to run, for around 20 minutes in all. The output of the command will include a number of tables:

```
 Name                    Time R^2   Time/Run             95ci
--------------------- ---------- ---------- -----------------
 parts_sexp:262144         1.00    10.18ms   -0.03ms +0.04ms
 parts_sexp:524288         1.00    20.39ms   -0.14ms +0.22ms
 parts_sexp:786432         1.00    30.32ms   -0.20ms +0.16ms
 ...
 fused_sexp:262144         1.00     2.01ms   -0.01ms +0.01ms
 fused_sexp:524288         1.00     3.99ms   -0.01ms +0.01ms
 fused_sexp:786432         1.00     6.01ms   -0.02ms +0.02ms
```

and the results will also be collected into csv files in the directory `/home/opam/flap/paper/csv/`.

If you encounter the error message

```
Error: index sets are not consistent.  Try increasing QUOTA (e.g. QUOTA=20 make bench)
```

then the benchmarks have not completed successfully and you should re-run them with a sufficiently large `QUOTA` number, e.g.:

```
QUOTA=20 make bench
```

to ensure that the benchmarks run for long enough to produce enough samples for the statistical analysis that they use.

You can verify the paper's claim by examining the numbers in the tables. For each benchmark, for each input size, the `fused` implementation should have the lowest running time. For example, in the table above, the time for the `fused` implementation on input size `786432` is `6.01ms`, while the time for the `parts` implementation is `30.32`.

We also provide a script `throughput.py` that calculates throughputs (as shown in Figure 11) from the timings recorded by `make bench`. Running `throughput.py` in the `/home/opam/flap` directory will compute a CSV table with the throughput times for each benchmark and implementation:

```
benchmark,fused,staged,ocamlyacc,...
json,1359.485981308411,168.69092947293146,236.19560510933104,...
sexp,212.6929006085193,92.14200351493848,76.37115804806992,...
arith,56.54328478964402,29.315226510067113,29.63008762012436,...
pgn,285.69604189639125,81.13024588657353,67.271636669281,...
ppm,103.6371188192452,26.605863127563445,15.679888979107007,...
csv,322.7958054219004,0,69.96887094060239,76.48532552766736,...
```

Both the relative and absolute results depend heavily on the system on which the benchmarks are run, so it is very unlikely that the numbers you see will correspond directly to the numbers in the paper. (There is even a small possibility that in some cases the `fused` implementation will not perform as well as the other implementations, but we have not observed that on the various systems on which we have run our evaluation.)

Claim 13: Lexer implementation in the evaluation (5 minutes)

The paper claims that the benchmark implementations use either ocamllex or combinators:

> For lexing we use ocamllex for (a)–(c), and the combinators supplied by each library for (d)–(f). Imple-

You can verify this claim by examining the benchmark code. For example, for the json benchmark, the first three benchmarks use ocamllex for lexing (indicated by the mll file extension):

```
json_lexer.mll                      # OCamlyacc lexer
json_lexer_menhir_code.mll          # Menhir lexer (code)
json_lexer_menhir_table.mll         # Menhir lexer (table)
```

while the ParTS and asp implementations use the combinators supplied with those systems:

```
json_parts.ml                       # ParTS lexer+parser
json_staged_combinator_parser.ml    # Asp lexer+parser
```

The ParTS code in our repository is the generated code distributed by the ParTS implementers. The asp code is the source for the asp benchmarks, and includes the lexer specification:

From /home/opam/flap/benchmarks/json/json_staged_combinator_parser.ml

```
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>.
```

The lexer implementation for the flap json benchmark is in /home/opam/flap/grammars/json_grammar.ml:

From /home/opam/flap/grammars/json_grammar.ml

```
let lexer =
  L.[
       chr '['             , P.Return LBRACKET;
       chr ']'             , P.Return RBRACKET;
       chr '{'             , P.Return LBRACE;
       chr '}'             , P.Return RBRACE;
       chr ','             , P.Return COMMA;
       chr ':'             , P.Return COLON;
       str "null"          , P.Return NULL;
       str "true"          , P.Return TRUE;
       str "false"         , P.Return FALSE;
       string              , P.Return STRING;
       decimal             , P.Return DECIMAL;
       charset "\r\n \t" , P.Skip;
  ]
```

Claim 14: Parser structure in the evaluation (10 minutes)

The paper claims that the evaluation uses identically-structured parsers for ocamlyacc and menhir and identically-structured parsers for ParTS, asp and flap:

> binators supplied by each library for (d)–(f). Implementations (a)–(c) use identically structured grammars (since menhir [Pottier and Régis-Gianas [n.d.]] accepts ocamlyacc files as input) and lexers based on ocamllex. Implementations (d)–(f) also use identically structured grammars, since they all use the standard parser combinator interface (§2.1). However, (d)–(f) use differently-

You can verify this claim by examining the benchmark code. For example, for the json benchmark you can run

```
ls -1   benchmarks/json/*parser*mly
```

to confirm that the parsers for the first three benchmarks are identical, since two of the files are symbolic links to the other:

```
-rw-r--r-- 1 root root 1.4K Mar  7 17:16 benchmarks/json/json_parser.mly
lrwxrwxrwx 1 root root   15 Mar  7 17:16 benchmarks/json/json_parser_menhir_code.mly -> json_parser.mly
lrwxrwxrwx 1 root root   15 Mar  7 17:16 benchmarks/json/json_parser_menhir_table.mly -> json_parser.mly
```

The implementation of the asp json parser is given in benchmarks/json/json_staged_combinator_parser.ml. Here is an excerpt:

From /home/opam/flap/benchmarks/json/json_staged_combinator_parser.ml

```
let value = fix @@ fun value ->
  let member = tok STRING >>>
               maybe (tok COLON >>> value) $
               fun p -> .< match .~p with  (_,None) -> 1
                                          | (_,Some(_,v)) -> 1 + v >. in
  let obj = delim LBRACE (commasep member) RBRACE
  and arr = delim LBRACKET (commasep value) RBRACKET
```

The implementation of the flap json parser is given in grammars/json_grammar.ml. Here is the corresponding excerpt:

From /home/opam/flap/grammars/json_grammar.ml

```
let value = P.(fix @@ fun value ->
  let member = string_ >>>
               option (colon >>> value) $
               fun p -> let_ p @@ fun p -> inj .< match .~(dyn p) with  (_,None) -> 1
                                                                       | (_,Some(_,v)) -> 1 + v >. in
  let obj = delim lbrace (commasep member) rbrace
  and arr = delim lbracket (commasep value) rbracket
```

Our artifact includes only the source (not the generated code) for the ParTS implementation. The source is available at https://github.com/draperlaboratory/parts/blob/master/theories/Json.v. The report that describes ParTS says that the benchmarks are re-implementations of the implementations in asp:

> Second, we evaluated the library's performance by re-implementing two of the key benchmarks from the TAAP paper (s-expressions and JSON).
>
> *ParTS: Final Report* (Chris Casinghino & Cody Roux)

Claim 15: flap has linear-time parsing (15–30 minutes)

The paper claims that flap and the other implementations have linear-time parsing:

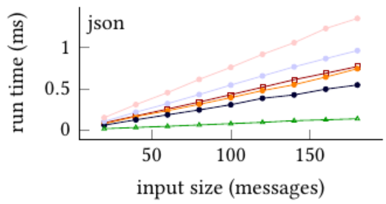

Fig. 12. Linear-time parsing (colors as Fig. 11)

You can verify this claim by examining the ratios between input sizes and running times in the figures reported by make bench. For example, in the following numbers for the sexp benchmark

```
fused_sexp:262144          1.00      2.01ms    -0.01ms +0.01ms
fused_sexp:524288          1.00      3.99ms    -0.01ms +0.01ms
fused_sexp:786432          1.00      6.01ms    -0.02ms +0.02ms
```

as the input size doubles from `262144` to `524288` and triples to `786432`, the running time also doubles from `2ms` to `4ms` and triples to `6ms`.

The ratios will not always be perfectly exact, but they should show an approximately linear relationship.

Claim 16: Our evaluation is based on six benchmarks (10 minutes)

The paper claims that the evaluation is based on six benchmarks:

The benchmarks are largely taken from Krishnaswami and Yallop [2019] (using the same test corpora), except for the CSV benchmark (which uses a set of files of various sizes and dimensions, using a random variety of textual and numeric data). They are:

(1) (pgn) Parse 6759 Portable Game Notation chess game descriptions, and extract game results.
(2) (ppm) Parse and check semantic properties (e.g. pixel count, color range) of Netpbm files.
(3) (sexp) Parse S-expressions with alphanumeric atoms, returning the atom count.
(4) (csv) Parse CSV files (Shafranovich [2005], with mandatory terminating CRLF), checking row lengths. This benchmark has no asp implementation, because distinguishing escaped double-quotes "" from unescaped quotes " in the lexer needs multiple characters of lookahead.
(5) (json) Parse JSON using the grammar by Jonnalagedda et al. [2014], returning the object count.
(6) (arith) Parse and evaluate terms in a mini language (arithmetic/comparison/binding/branching).

You can verify this claim by examining the six subdirectories of the `/home/opam/flap/benchmarks` directory (excluding `common`), which correspond to the six benchmarks described in the paper.

Claim 17: `flap` outputs have the reported sizes (10 minutes)

The paper claims that normalized grammars have certain sizes:

	Input		Normalized		Fused	Output
Grammar	Lex rules	CFEs	NTs	Prods	Prods	Functions
pgn	13	95	38	53	91	206
ppm	6	10	5	6	16	55
sexp	4	11	3	6	9	11
csv	3	14	5	7	7	20
json	12	42	9	33	42	97
arith	14	143	28	55	83	209

Table 1. Sizes of inputs, intermediate forms, and generated code

However, measurements largely dispel these concerns. Table 1 lists parser representation sizes at various stages in `flap`'s pipeline. The leftmost columns show the size of the input parsers, measured as the number of lexer rules (both **Return** and **Skip**) and the number of CFE nodes, as described in Fig. 3a. The central columns show the number of nonterminals and productions after conversion to DGNF using the procedure in §3; they show that normalization for typed CFEs does not produce the drastic increases in size that occur in the more general conversion to GNF. The next column to the right shows the grammar size after fusion (§4). Fusion does not alter the number of nonterminals, but can add productions; for example, the **Skip** rules in the sexp lexer add additional productions to each nonterminal. Finally, the rightmost column shows the number of function bindings in the code generated by `flap`. Comparing this generated function count with the number of CFEs in the input reveals an unalarming relationship: with one exception (ppm), their ratio barely exceeds 2.

You can verify this claim by loading the example grammars into the MetaOCaml top level. For example, for the `sexp` grammar, the following sequence of commands will load the dependencies and then compile the grammar:

```
#use "topfind";;
#require "flap";;
#require "ppx_deriving.std";;
#mod_use "grammars/grammars_common.ml";;
#mod_use "grammars/sexp_grammar.ml";;
flush stderr;;
```

The output should include the following lines:

```
4 lexing rules
11 context-free expressions
3 normalized nonterminals
6 normalized productions
9 fused productions
```

The number of generated functions is not reported, but you can verify the figures in Table 1 by examining the generated code. For example, to print out the generated code for the `sexp` grammar, type the following at the MetaOCaml top level after executing the commands above:

```
Sexp_grammar.code;;
```

(Note that the figures in the table refer to the top-level mutually-recursive function group (i.e. the functions bound with `let rec ... and ...`), not to locally bound variables. Note also that Table 1 is slightly out of date, and several grammars now produce a smaller number of functions: `Pgn_grammar.code` contains 203 functions (not 206); `Csv_grammar.code` contains 17 functions (not 20); `Json_grammar.code` contains 93 functions (not 97).)

Claim 18: `flap` compilation time is acceptable (5 minutes)

Compilation time (ms)
212
3.60
0.331
0.499
28.5
460

Table 2. Compilation time (type-checking, normalization, fusion, code generation)

Table 2 shows the compilation time for the benchmark grammars. For each, the total time taken to type-check and normalize the grammar, fuse the grammar and lexer and generate code is below half a second.

You can verify this claim by compiling and running the `flap` tests, since the tests report the compilation time for each benchmark grammar:

```
cd /home/opam/flap/
dune runtest -f
```

The output should include a report of the following form[1]:

```
[sexp]: compilation time : 0.331ms
[pgn]: compilation time : 212ms
[ppm]: compilation time : 3.60ms
[json]: compilation time : 28.5ms
[csv]: compilation time : 0.499ms
[intexp]: compilation time : 460ms
```

The exact compilation time will vary by the system used for testing, but none of the parsers should take more than around two seconds to compile, unless the system is extremely slow.

[1] The name `intexp`, which is also used in various parts of the source, corresponds to the benchmark named `arith` in the paper.

Claims not supported by the artifact

§3 of the paper presents some metatheoretical results related to grammar normalization:

Theorem 3.1 (Deterministic Parsing). *If G is a DGNF grammar, then for any expansion $G \vdash n \rightsquigarrow w$, there is a unique derivation for this expansion.*

Lemma 3.2 (Productions of Null). *Given $\Gamma; \Delta \vdash g : \tau$ and $\mathcal{N}[\![\, g \,]\!]$ returns $n \Rightarrow G$, we have $\tau.\textsc{Null} = \mathsf{true}$ if and only if (1) $n \longrightarrow \epsilon \in G$; or (2) $n \longrightarrow \alpha \in G$ where $(\alpha : \tau') \in \Gamma$ and $\tau'.\textsc{Null} = \mathsf{true}$. In other words, if $\tau.\textsc{Null} = \mathsf{false}$, then $n \longrightarrow \epsilon \notin G$.*

Theorem 3.3 (Well-definedness). *If $\Gamma; \Delta \vdash g : \tau$, then $\mathcal{N}[\![\, g \,]\!]$ returns $n \Rightarrow G$ for some G and n.*

Lemma 3.4 (Internal normal form). *Given $\Gamma; \Delta \vdash g : \tau$ and $\mathcal{N}[\![\, g \,]\!]$ returns $n \Rightarrow G$,*

- *if $(n \longrightarrow \alpha\ \overline{n}) \in G$, then $\alpha \in \mathrm{dom}\,(\Gamma)$;*
- *if $(n' \longrightarrow \alpha\ \overline{n}) \in G$ for any n', then $\alpha \in \mathrm{fv}\,(g)$, and thus $\alpha \in \mathrm{dom}\,(\Gamma, \Delta)$.*

Corollary 3.5 (Normalizing without internal normal form). *Given $\bullet; \bullet \vdash g : \tau$, if $\mathcal{N}[\![\, g \,]\!]$ returns $n' \Rightarrow G$, then any production in G is either $n \longrightarrow \epsilon$ or $n \longrightarrow t\ \overline{n}$ for some n, t and $\overline{n}$.*

Lemma 3.6 (Terminals in First). *Given $\Gamma; \Delta \vdash g : \tau$ and $\mathcal{N}[\![\, g \,]\!]$ returns $n \Rightarrow G$, we have $t \in \tau.\textsc{First}$ if and only if (1) $(n \longrightarrow t\ \overline{n}) \in G$; or (2) $(n \longrightarrow \alpha\ \overline{n}) \in G$ where $(\alpha : \tau') \in \Gamma$ and $t \in \tau'.\textsc{First}$.*

Theorem 3.7 ($\mathcal{N}[\![\, g \,]\!]$ produces DGNF). *If $\bullet; \bullet \vdash g : \tau$, then $\mathcal{N}[\![\, g \,]\!]$ returns $n \Rightarrow D$ for some n, D.*

Theorem 3.8 (Soundness). *Given $\bullet; \bullet \vdash g : \tau$ and $\mathcal{N}[\![\, g \,]\!]$ returns $n \Rightarrow G$, we have $w \in [\![g]\!]_{\bullet}$ if and only if $G \vdash n \rightsquigarrow w$ for any w.*

Since these are claims about our formalisation, not about our implementation, the artifact does not contain evidence for them. Instead, their proofs are given in an appendix to the paper.