Fusing Lexing and Parsing

Anonymous Author(s)

Abstract

Lexers and parsers are typically defined separately and connected by a token stream. This separate definition is important for modularity, but harmful for performance.

We show how to *fuse* together separately-defined lexers and parsers, drastically improving performance without compromising modularity. Our staged parser combinator library, `flap`, provides a standard parser combinator interface, but generates specialized token-free code that runs several times faster than `ocamlyacc` on a range of benchmarks.

Keywords: parsing, multi-stage programming, optimization, fusion

1 Introduction

Software systems are easiest to understand when their components have clear interfaces that hide internal details. For example, a typical compiler includes separate lexer and parser components that communicate via a token stream.

Unfortunately, while interfaces improve clarity, they can harm performance, since hiding internal details reduces optimization opportunities. Parsers exemplify this tension: the token stream interface isolates parser definitions from character syntax details like whitespace, but it also carries overheads that reduce parsing speed.

Parsers built for efficiency avoid backtracking: only the initial token of the stream is typically needed at any time. However, even with this restriction, materializing and case-switching on tokens comes with a cost, and eliminating tokens altogether from generated code can drastically improve performance.

1.1 Contributions

This paper introduces a transformation that entirely eliminates tokens by fusing together lexers and parsers built using standard tools. The next section summarizes these tools: Brzozowski's derivatives [3] for lexers (as reformulated by Owens, Reppy and Turon [21]), and Krishnaswami and Yallop's typed algebraic parser combinators [17]. The three subsequent sections present contributions:

- Section 3 presents (and proves correct) a translation from typed context-free expressions into a variant of Greibach Normal Form for deterministic languages that simplifies fusion.
- Section 4 presents lexer-parser fusion, showing how to transform a separately-defined lexer and parser into a single piece of code that is specialized for calling contexts, entirely avoids materializing tokens, and case-switches only on individual characters, not on intermediate structures.
- Section 5 shows that fusing lexers and parsers results in code that runs four to six times faster than code produced by *ocamllex* and *ocamlyacc*, and assesses other metrics, such as code size.

Finally, Section 6 surveys related work and Section 7 sets out some directions for further development.

2 Background: lexer and parser combinators

We present lexer-parser fusion using a parser combinator library, `flap` (***f**used **l**exing **a**nd **p**arsing*). As Figure 1 shows, `flap`'s interface is built from standard lexer and parser combinators drawn from existing work [17, 21]. This section gives an overview of those combinators; Sections 3 and 4 then present `flap`'s novel code generation architecture, which normalizes grammars built from the parsing combinators, fuses the grammar together with specializations of the lexer into a single representation, and then uses multi-stage programming to generate efficient token-free code from that representation.

2.1 Derivatives of regular expressions

Since lexical syntax is typically regular, lexers are typically defined using regular expressions (regexes). One particularly elegant formulation of regex matching, introduced almost six decades ago [3], is based on the idea of *derivatives*.

The *derivative* of a regex r with respect to a character c is another regex $\partial_c\ r$ that matches $c \cdot s$ exactly when r matches s. For example, the regex $(b|c)+$ matches a sequence of one of one or more occurrences of b and c in any order. For a string that begins with c, either an empty suffix or some further sequence of b and c is acceptable, and so we have:

$$\partial_c\ (b|c)+ \quad = \quad (b|c)*$$

The full rules, shown in Figure 2, are defined inductively on the syntax of regexes, with cases for the standard constructs $\bot$ (which matches nothing), ϵ (which matches only the empty string), characters b and c, sequencing, alternation, and Kleene star, and for the less commonly-supported constructs intersection and negation. (The *nullability* function $\nu(r)$ in the sequencing rule expands to ϵ if r matches ϵ and $\bot$ otherwise). We refer the reader to Owens et al. [21] for a fuller exposition.

Using derivatives, it is straightforward to construct an automaton from a regular expression, taking regular expressions r as states, and adding a transition from r_i to r_j via

PL'18, January 01–03, 2018, New York, NY, USA
2018.

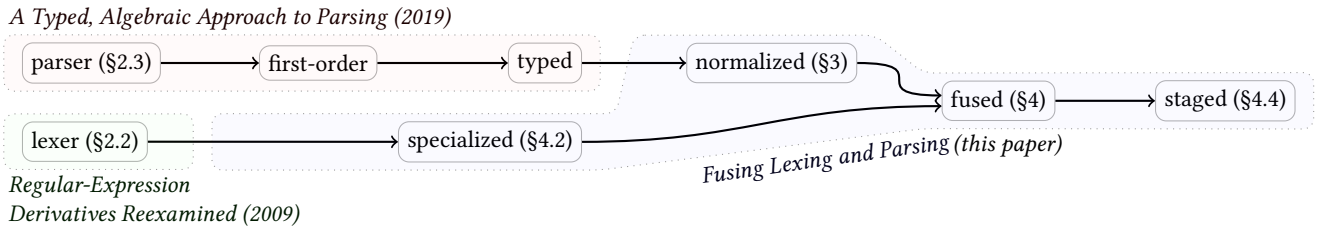
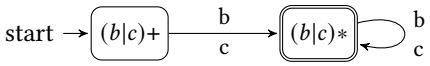


Figure 1. Architecture of `flap`

$$
\begin{array}{rcl}
\partial_c\ \bot & = & \bot \\
\partial_c\ \epsilon & = & \bot \\
\partial_c\ c & = & \epsilon \\
\partial_c\ b & = & \bot
\end{array}
\qquad
\begin{array}{rcl}
\partial_c\ (r \cdot s) & = & \partial_c\ r \cdot s \mid \nu(r) \cdot \partial_c\ s \\
\partial_c\ (r \mid s) & = & \partial_c\ r \mid \partial_c\ s \\
\partial_c\ r* & = & \partial_c\ r \cdot r* \\
\partial_c\ (r\ \&\ s) & = & \partial_c\ r\ \&\ \partial_c\ s \\
\partial_c\ \neg r & = & \neg(\partial_c\ r)
\end{array}
$$

Figure 2. Derivatives of regular expressions

character c whenever $\partial_c\ r_i = r_j$. For example, here is an automaton for $(b|c)+$:



In this example, the transitions all target the same state, since $\partial_b\ (b|c)+ = \partial_c\ (b|c)+ = \partial_b\ (b|c)* = \partial_c\ (b|c)* = (b|c)*$. Additionally, since that state also accepts the empty string, it is marked as an accepting state for the whole automaton.

Constructing automata is a common way to implement regex matchers, and derivatives make it straightforward to built automata that are deterministic and compact. However, an even simpler approach to constructing matches using derivatives, used in `flap`, is to use *multi-stage programming* [27] to generate code directly.

Regex matching is an archetypal example of staged computation [7] as found in languages like BER MetaOCaml [14]: although matching is a function of two inputs, regex and string, since the former is typically available first, it can be used to construct specialized code for processing the latter. In other words, while an unstaged matcher might have the following two-argument OCaml type

```
val matchr : regex → string → bool
```

a staged matcher is instead a function of one argument which generates code for another function of one argument:

```
val smatchr : regex → (string → bool) code
```

For the regex $(b|c)+$, `smatchr` might generate code for two mutually-recursive functions for the two regexes $(b|c)+$ and $(b|c)*$ found in the corresponding automaton:

```
≪ let rec bcplus s i = (i <> length s) &&
    match s.[i] with 'b'|'c' → bcstar s (i+1)
                   | _ → false
  and bcstar s i = (i = length s) ||
```

$$
\begin{array}{rcl}
\text{id} & \Rightarrow & \textbf{Return}\ \textsc{atom} \\
\text{space} & \Rightarrow & \textbf{Skip} \\
(& \Rightarrow & \textbf{Return}\ \textsc{lpar} \\
) & \Rightarrow & \textbf{Return}\ \textsc{rpar}
\end{array}
\qquad
\begin{array}{rcl}
\text{id} & \stackrel{\text{def}}{=} & \texttt{[a-z]+} \\
\text{space} & \stackrel{\text{def}}{=} & \texttt{␣ | \textbackslash n}
\end{array}
$$

Figure 3. S-expression lexer

```
    match s.[i] with 'b'..'c' → bcstar s (i+1)
                   | _ → false
  in fun s → bcplus s 0 ≫
```

The function for each regex r and string s follows a simple pattern, returning **`true`** if r is nullable and s is empty, and moving to the function for $\partial_c\ r$ if s begins with c.

The quotation notation $\ll e \gg$ in this example indicates a typed code value; it is one of MetaOCaml's two fundamental constructs (along with antiquotation $\tilde{e}$) for staging. Generating mutually-recursive definition requires the use of a third construct, *letrec-generation* [32], which Section 4.4.2 describes in more detail.

2.2 Lexing with derivatives

Regexes built from derivatives are convenient for building lexers. A lexer is typically defined as an ordered mapping from regular expressions to *actions*, where an action might return a token, raise an error or invoke the lexer recursively to skip over some input. Figure 3 gives an example lexer with four actions: three of which return tokens atom, lpar and rpar, and one of which skips over whitespace.

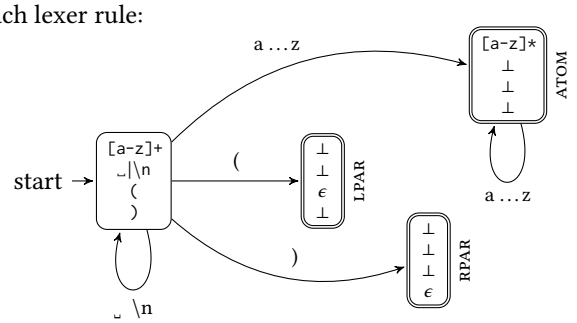
Using MetaOCaml it is straightforward to express lexers as functions that accept lists of regex-action pairs and return code:

```
type 'a action =
 Skip | Error of string | Return of 'a code
val slex :
 (re * 'a action) list →(string → 'a) code
```

For succinctness, our exposition generally omits the treatment of `Error` actions (which is entirely straightforward).

Regex derivatives extend naturally to lexers by matching the input string against multiple regexes in parallel. Here is the automaton for matching one token with the sexp lexer,

where each state corresponds to a vector of regexes, one for each lexer rule:



The transition function ∂_c acts pointwise on the regex vector. **Return** rules correspond to labeled accepting states, and the **Skip** rule resets the vector to its initial state.

As Owens et al. [21] show, lexers based on derivatives provide a practical basis for real-world lexing tools such as ml-ulex and PLT Scheme. One particularly useful feature for implementing lexers is the support derivatives provide for negation and disjunction, which make it straightforward to transform implicitly-ordered clauses for regexes r and s into order-independent disjoint clauses for r and $\neg(r)\&s$.

2.3 Parsing with typed context-free expressions

Parser combinators, introduced almost four decades ago by Wadler [30], provide an elegant way to define parsers using functions. A parser combinator library provides functions denoting token-matching, sequencing, disjunction, and so on, allowing the library user to describe a parser by combining these functions in a way that reflects the structure of the corresponding grammar.

Figure 4 shows partial interfaces for constructing both regexes (type `re`) and parsers (type `pa`) in this way. Both interfaces provide functions for token matching, sequencing and recursion. However, there are some important differences: first, regexes act on characters, while parsers act on tokens (type `tok`, a parameter of the library); second, parsers provide a general-purpose recursion operator `fix`, while regexes offer only the more restrictive Kleene `star`; finally, the parser type is parameterized, allowing parsers to construct and return suitably-typed syntax trees.

The earliest parser combinator libraries represented nondeterministic parsers, with support for arbitrary backtracking and multiple results. Parsers defined in this way enjoyed various pleasant properties (such as a rich equational theory), but suffered from potentially disastrous performance. In a recent departure from the nondeterministic tradition, Krishnaswami and Yallop [17] define *typed context-free expressions*, whose types track properties such as FIRST sets and nullability in order to preclude nondeterminism and ensure linear-time parsing using a single token of lookahead. Krishnaswami and Yallop's design provides the standard set of parser combinators (Figure 4), but adds an additional type-checking step. They further apply multi-stage programming to ensure that type-checking is completed before parsing begins, and to generate specialized parsing code based on type information, leading to performance competitive with `ocamlyacc`.

Figures 5 and 6 summarise Krishnaswami and Yallop's type system. A type is a triple recording nullability, the first set, and FLAST (analogous to the FOLLOW set). There is one typing rule for each combinator (e.g. sequencing $g \cdot g'$ and recursion $\mu x : \tau.g$); types are constructed using corresponding combinators (e.g. $\tau_1 \cdot \tau_2$). The two contexts Γ and Δ restrict the positions in which variables can occur to disallow left recursion, and the side conditions $\tau_1 \circledast \tau_2$ and $\tau_1 \ \#\ \tau_2$ on the rules for sequencing and alternation reject ambiguous grammars, ensuring respectively that strings matched by sequenced parsers have a unique decomposition, and that the languages matched by alternated parsers do not overlap.

The following example shows how to use Krishnaswami and Yallop's parsers to define a grammar for s-expressions, using a token type with LPAR, RPAR and ATOM constructors:

```
fix (fun e → (tok LPAR >>> star e >>> tok RPAR)
            $ fun p → ≪ Sexp (snd (fst ~p)) ≫
          <|> tok ATOM $ fun s → ≪ Atom ~s ≫)
```

The `fix`, `tok`, `>>>` and `star` constructors are shown in Figure 4. The example additionally uses alternation `<|>` and the map function `$`, which transforms the result of parsing via a user-defined function.

3 Normalizing context-free expressions

The algorithm for parsing with typed context-free expressions introduced by Krishnaswami and Yallop is efficient at a high level, since it uses only a single token of lookahead and its execution time is linear in the length of its input. However, it is less efficient at a low level, since it examines each token multiple times: once at each alternation $g \vee g'$ in the grammar, and then once again at the token combinator c.

Krishnaswami and Yallop address these low-level inefficiencies using a variety of multi-stage programming techniques, ultimate achieving performance that is competitive with ocamllex and ocamlyacc. Here we take a more systematic approach, making use of the guarantees offered by the types to transform grammars into a normal form that is amenable to further optimization.

More precisely, our normal form gathers together the places in the grammar that involve branching on tokens, allowing tokens to be discarded immediately after inspection, and ultimately eliminating the need to materialize tokens altogether (Section 4). As we shall see, these optimizations make parsers built from typed context-free expressions significantly more efficient than both `ocamlyacc` and Krishnaswami and Yallop's system (Section 5).

As an example, consider the s-expression grammar from Section 2.3, written here using the succinct mathematical

```
(* Regex combinators *)
type re
val chr: char → re          (* token match *)
val (>>>): re → re → re      (* sequence  *)
val star: re → re            (* recursion *)
...

(* Parser combinators *)
type 'a pa
val tok: 'a tok → 'a pa
val (>>>): 'a pa → 'b pa → ('a * 'b) pa
val fix: ('a pa → 'a pa) → 'a pa
...
```

Figure 4. Regex and parser combinators

$$\frac{}{\Gamma;\Delta \vdash \epsilon : \tau_\epsilon} \qquad \frac{}{\Gamma;\Delta \vdash c : \tau_c} \qquad \frac{}{\Gamma;\Delta \vdash \bot : \tau_\bot}$$

$$\frac{x : \tau \in \Gamma}{\Gamma;\Delta \vdash x : \tau} \qquad \frac{\Gamma;\Delta, x : \tau \vdash g : \tau}{\Gamma;\Delta \vdash \mu x : \tau . g : \tau}$$

$$\frac{\Gamma;\Delta \vdash g : \tau \qquad \Gamma,\Delta;\bullet \vdash g' : \tau' \qquad \tau \circledast \tau'}{\Gamma;\Delta \vdash g \cdot g' : \tau \cdot \tau'}$$

$$\frac{\Gamma;\Delta \vdash g : \tau \qquad \Gamma;\Delta \vdash g' : \tau' \qquad \tau \# \tau'}{\Gamma;\Delta \vdash g \vee g' : \tau \vee \tau'}$$

Figure 5. Typing for context-free expressions

$$\text{Types} \quad \tau \in \{\textsc{Null} : 2;\ \textsc{First} : \mathcal{P}(\Sigma);\ \textsc{FLast} : \mathcal{P}(\Sigma)\}$$

$$\begin{aligned}
\tau_1 \circledast \tau_2 &\stackrel{\text{def}}{=} \tau_1.\textsc{FLast} \cap \tau_2.\textsc{First} = \emptyset \wedge \neg\tau_1.\textsc{Null} \\
\tau_1 \,\#\, \tau_2 &\stackrel{\text{def}}{=} (\tau_1.\textsc{First} \cap \tau_2.\textsc{First} = \emptyset) \wedge \neg(\tau_1.\textsc{Null} \wedge \tau_2.\textsc{Null}) \\
b \Rightarrow S &\stackrel{\text{def}}{=} \text{if } b \text{ then } S \text{ else } \emptyset
\end{aligned}$$

$$\begin{aligned}
\tau_\epsilon &= \{\textsc{Null} = \text{true};\ \textsc{First} = \emptyset;\ \textsc{FLast} = \emptyset\} \\
\tau_c &= \{\textsc{Null} = \text{false};\ \textsc{First} = \{c\};\ \textsc{FLast} = \emptyset\} \\
\tau_\bot &= \{\textsc{Null} = \text{false};\ \textsc{First} = \emptyset;\ \textsc{FLast} = \emptyset\} \\
\tau_1 \cdot \tau_2 &= \begin{cases} \textsc{Null} = \tau_1.\textsc{Null} \wedge \tau_2.\textsc{Null} \\ \textsc{First} = \tau_1.\textsc{First} \cup \tau_1.\textsc{Null} \Rightarrow \tau_2.\textsc{First} \\ \textsc{FLast} = \tau_2.\textsc{FLast} \cup \tau_2.\textsc{Null} \Rightarrow (\tau_2.\textsc{First} \cup \tau_1.\textsc{FLast}) \end{cases} \\
\tau_1 \vee \tau_2 &= \begin{cases} \textsc{Null} = \tau_1.\textsc{Null} \vee \tau_2.\textsc{Null} \\ \textsc{First} = \tau_1.\textsc{First} \cup \tau_2.\textsc{First} \\ \textsc{FLast} = \tau_1.\textsc{FLast} \cup \tau_2.\textsc{FLast} \end{cases}
\end{aligned}$$

Figure 6. Types for context-free expressions

notation for context-free expressions rather than OCaml syntax, and with the Kleene star expanded to an explicit fixed point:

$$\mu\ \text{sexp}\ .(\textsc{lpar} \cdot (\mu\ \text{sexps}\ .\ \epsilon \vee\ \text{sexp}\ \cdot\ \text{sexps}\ ) \cdot \textsc{rpar}) \vee \textsc{atom}$$

The type system of Figure 5 ensures that parsers built from this grammar can branch with a single token of lookahead. However, some reasoning is needed to determine exactly how to branch. For instance, in the sub-grammar $\epsilon \vee$ sexp $\cdot$ sexps, neither alternative explicitly matches a token. Determining which branch to take therefore requires analysing the types to calculate whether the next tokens in the input fall into the First set of either ϵ or sexp $\cdot$ sexps. Furthermore, once the token has been examined and the branch taken, the parsing algorithm must examine it a second time. For example, since $\textsc{lpar} \in \textsc{First}(\text{sexp} \cdot\ \text{sexps})$, the token lpar causes the parsing algorithm to switch to the sexp $\cdot$ sexps branch. After the switch, parsing again uses the typing information for the selected branch to determine which of its sub-branches to take (i.e. (lpar ...) or atom). Eventually, the parsing algorithm encounters the lpar node in the grammar, and the token is consumed.

The normalization algorithm in this section transforms grammars into a form that avoids the need for repeated branching. Here is the result of normalizing the s-expression grammar, given in BNF form:

$$\begin{aligned}
\text{sexp} &::= \textsc{lpar}\ \text{sexps}\ \textsc{rpar} \\
&\mid \textsc{atom} \\
\text{rpar} &::= \textsc{rpar} \\
\text{sexps} &::= \textsc{lpar}\ \text{sexps}\ \textsc{rpar}\ \text{sexps} \\
&\mid \textsc{atom}\ \text{sexps} \\
&\mid \epsilon
\end{aligned}$$

With this normalized form, parsing a sexp involves reading the next token, and branching to the first, second or third branch depending on whether the token is lpar, atom or something else. In the first two cases the token is consumed immediately, and parsing moves on to the next token in the input. The last case is somewhat more costly, since the token may be needed again immediately afterwards, and more care is needed to avoid wasted work.

3.1 Deterministic Greibach Normal Form

Readers familiar with the theory of formal languages might recognise that the normalized s-expression grammar is (almost) in *Greibach Normal Form* (GNF) [11]. In GNF, all the productions of a grammar take the form $A \longrightarrow cB_1B_2 \ldots B_k (k \geq 0)$ where A and B_i are nonterminals and c is a terminal.

The form defined here is a variant of GNF, which we call *deterministic GNF*, with two key differences. First, we require that, for any pair of a nonterminal A and a terminal c, there

$$
\begin{array}{llll}
\text{context-free expr} & g & ::= & \epsilon \mid \bot \mid c \mid \alpha \mid g_1 \vee g_2 \\
 & & & \mid g_1 \cdot g_2 \mid \mu\alpha.\, g \\
\text{terminal} & t & & \\
\text{non-terminal} & n & & \\
\text{normal form} & N & ::= & t\,\overline{n} \mid \epsilon \mid n_\alpha\,\overline{n} \\
\text{grammar} & G & ::= & \overline{n_i \to N_i}^{\,i} \\
\text{environment} & \phi & ::= & \bullet \mid \phi, \alpha \to n_\alpha
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{fv}(\epsilon) & = & \emptyset \\
\mathsf{fv}(\bot) & = & \emptyset \\
\mathsf{fv}(c) & = & \emptyset \\
\mathsf{fv}(g_1 \cdot g_2) & = & \mathsf{fv}(g_1) \cup \mathsf{fv}(g_2) \\
\mathsf{fv}(g_1 \vee g_2) & = & \mathsf{fv}(g_1) \cup \mathsf{fv}(g_2) \\
\mathsf{fv}(\mu\alpha.\, g) & = & \mathsf{fv}(g) - \{\alpha\} \\
\mathsf{fv}(\alpha) & = & \{\alpha\}
\end{array}
$$

$$
\begin{array}{lcll}
\mathcal{N}(\bot)_\phi & = & (\emptyset,\ n) & (\text{fresh } n) \\
\mathcal{N}(\epsilon)_\phi & = & (n \to \epsilon,\ n) & \\
\mathcal{N}(c)_\phi & = & (n \to c,\ n) & \\
\mathcal{N}(g_1 \cdot g_2)_\phi & = & (\overline{n \to N_i\, n_2}^{\,i} \cup G_1 \cup G_2,\ n) & \\
 & & \text{where} \quad \mathcal{N}(g_1)_\phi = (G_1,\ n_1) & \\
 & & \qquad\quad \mathcal{N}(g_2)_\phi = (G_2,\ n_2) & \\
 & & \qquad\quad G_1.n_1 = \overline{n_1 \to N_i}^{\,i} & \\
\mathcal{N}(g_1 \vee g_2)_\phi & = & (\overline{n \to N_i}^{\,i} \cup \overline{n \to N_j}^{\,j} \cup G_1 \cup G_2, n) & \\
 & & \text{where} \quad \mathcal{N}(g_1)_\phi = (G_1,\ n_1) & \\
 & & \qquad\quad \mathcal{N}(g_2)_\phi = (G_2,\ n_2) & \\
 & & \qquad\quad G_1.n_1 = \overline{n_1 \to N_i}^{\,i} & \\
 & & \qquad\quad G_2.n_2 = \overline{n_2 \to N_j}^{\,j} & \\
\mathcal{N}(\mu\alpha.\, g)_\phi & = & (\overline{n_\alpha \to N_i}^{\,i} \cup \overline{\overline{n_j \to N_i\,\overline{n}_j}^{\,i}}^{\,j} \cup G',\ n_\alpha) & \\
 & & \text{where} \quad (G,\ n) = \mathcal{N}(g)_{\phi, \alpha \to n_\alpha} & \\
 & & \qquad\quad G.n = \overline{n \to N_i}^{\,i} & \\
 & & \qquad\quad G = \overline{n_j \to n_\alpha\,\overline{n}_j}^{\,j} \cup G' & \\
 & & \qquad\quad n' \to n_\alpha\,\overline{n}' \notin G' \text{ for all } n' & \\
\mathcal{N}(\alpha)_\phi & = & (n \to \phi(\alpha),\ n) &
\end{array}
$$

Figure 7. Normalization of typed context-free expressions

is at most one production beginning $A \to c \ldots$. Second, we allow each nonterminal to have an optional ϵ-production $A \to \epsilon$, with the proviso that the production may only be used when no terminal symbol in the non-ϵ productions for A matches the input string. (Section 3.1.1 explains this second difference in more detail.)

Taken together, these requirements ensure that grammars can be used for deterministic parsing with a single token of lookahead: that is, they capture syntactically those properties that Krishnaswami and Yallop enforce using types. Section 3.2 shows the translation from typed context-free expressions to grammars in deterministic GNF. Section 4 shows how `flap` makes use of these syntactic restrictions to implement lexer-parser fusion, eliminating tokens from generated code altogether.

3.1.1 Semantics of Deterministic GNF.

As mentioned above, the ϵ productions $A \to \epsilon$ in a deterministic GNF grammar carry the implicit restriction that they are only active when the terminal symbols in the other productions for A do not match the input string. More precisely, if the input string matches a branch $A \to cB_1B2 \ldots B_k$, then the branch (if exists) $A \to \epsilon$ cannot apply.

As an example, suppose that we want to generate the string LPAR LPAR RPAR RPAR. We start with

$$\text{sexp} \quad ::= \quad \text{LPAR sexps rpar}$$

Now, should we expand sexps to LPAR sexps rpar sexps or to ϵ? In a general nondeterministic grammar we can't tell just by examining the "input" string. But in this grammar we know that since the next symbol in the input is LPAR then the epsilon production does not apply, and we know that we must take the other branch.

As this example shows, these *guarded* epsilon productions complicate the definition and use of grammars. However, they have a simple practical motivation in parsers, where they represent an else branch that is taken if none of the active productive branches (i.e. productions with a terminal in leftmost position) matches the input.

Furthermore, like the other constraints on the grammar, guarded epsilon productions are a syntactic analogue of the constraints enforced by the types in Krishnaswami and Yallop's typed context-free expressions. Just like the types (and, in particular, the *separation* and *apartness* relations), guardedness ensures that there is never any ambiguity about whether an epsilon rule applies.

Finally, ϵ elimination in the normalization procedures for standard Greibach Normal Form has two deleterious effects: it introduces non-determinism, and it causes a substantial increase in the size of the grammar. In contrast, the analysis in Section 5.3 suggests that the size increase resulting from normalization to deterministic GNF in `flap` is relatively modest.

3.2 Normalization

Figure 7 defines a normalization algorithm for typed context-free expressions. The normalization function $\mathcal{N}(g)_\phi$ maps derivations $\Gamma; \Delta \vdash g : \tau$ to normalized grammars $(G,\ n)$, where G is a set of productions and n the distinguished start

nonterminal; the environment ϕ maps variables α found in the context-free expression to metavariables n_α, which are described in more detail below. Normalization is well-defined only for typed expressions; however, it does not scrutinise typing information and so we leave Γ, Δ and τ.

There are seven cases for the seven context-free expression constructors. Each case involves allocating a fresh nonterminal to use as start symbol. The cases with sub-expressions ($g_1 \cdot g_2$, $g_1 \vee g_2$ and $\mu\alpha.\ g$) are defined compositionally in terms of the normalization of those sub-expresions. Since normalization simply merges together all the production sets resulting from sub-expressions, the situation frequently arises where productions are not reachable from the start symbol; the definition here ignores this issue, since it is easy to trim unreachable productions in the implementation.

The three atomic cases, for $\bot$, ϵ and c, are straightforward. For $\bot$, normalization produces an empty grammar, with a start symbol and no productions. For each of ϵ and c, normalization produces a grammar with a single production whose right-hand side is ϵ or c respectively.

The cases for sequencing and alternation are defined compositionally in terms of the normalization of their sub-expressions. For sequencing $g_1 \cdot g_2$, normalization copies the productions for the start symbol of g_1, appending to each the start symbol n_2 of g_2. For alternation $g_1 \vee g_2$, normalization merges the productions for the start symbols n_1 and n_2 of g_1 and g_2 into the productions for the new start symbol n. The case for alternation is one of several places where the correctness of normalization depends on the types. The typing rules for alternation (Figure 5) depend on the apartness relation $\tau_1 \# \tau_2$ (Figure 6), which guarantees that the First sets of g_1 and g_2 do not intersect, and that at most one of g_1 and g_2 is nullable. Without that guarantee, the result of the normalization function might not be in deterministic GNF.

The final two cases deal with the binding fixed point operator $\mu\alpha.\ g$ and with bound variables α. Normalization for the fixed point operator $\mu\alpha.\ g$ takes place in two stages. First, the body g is normalized in an environment extended with an entry that associates α with a fresh metavariable n_α, which stands for the nonterminal of the fixed point. (The entry plays a similar role to the so-called chain rules $n_1 \longrightarrow n_2$ in Extended Greibach Normal Form [1].) The normalization result suggests that the grammar of the body g has a start symbol n. Then, according to the semantics of fixed point, we can proceed to tie the knot by copying the productions for n into the production for n_α. There is some extra work before we return the result. In particular, productions in G might start with n_α. While such form $(n_\alpha\ \overline{n}_j)$ is allowed by the syntax of N, our ultimate goal is to turn the productions into deterministic GNF, where every nonterminal either starts with a terminal or is ϵ. Now that we learn the productions of n_α, we look up and substitute in G all products that start with n_α. Note that we still allow n_α, as a special kind of nonterminal, to appear inside G if it is not the start of a production. As we will see, this effectively guarantees that normalizing closed context-free expressions produces deterministic GNF.

Normalization uses the environment to resolve variables α to metavariables n_α. In the case of α, we look up the metavariable n_α from the environment, and create a fresh start symbol which produces n_α. It may be tempting here to return n_α as a start symbol with no productions $(\emptyset,\ n_\alpha)$, but that would be wrong, as this does not create any grammar, losing the connection to n_α in the case of, e.g., normalizing $\alpha \vee g$ which tries to copy the productions from normalizing α.

3.3 Implementing normalization

The compositionality of the normalization algorithm simplifies the implementation of normalization in `flap`. Since the normalization of each term is defined in terms of the normal forms of its subterms, `flap` can represent terms in normal form. For example, if g and g' are `flap` parsers in normal form, then g `>>>` g' is also a parser in normal form, built from g and g' using the rules in Figure 7.

The simplicity of the normalization algorithm is reflected in the implementation of `flap`. Additionally, the most intricate part of the algorithm — dealing with fixed points — is also the subtlest part of the implementation. The implementation follows the formal algorithm closely, inserting placeholders that are tracked using an environment and resolved later. This kind of "backpatching" mirrors the way in which recursion is commonly implemented in eager functional languages such as OCaml [22]; if `flap` were instead implemented in a lazy language then it would be possible to implement fixed point normalization with significantly less fuss.

3.4 Properties of normalization

We establish the correctness of normalization via three properties: first, the result of normalization is a grammar in normal form; second, normalization succeeds for every well-typed grammar; third, normalization preserves semantics.

3.4.1 Normal Form. The first lemma says that the free variables in the normalization of a derivation of $\Gamma; \Delta \vdash g : \tau$ must be found in g.

Lemma 3.1. *If* $\Gamma; \Delta \vdash g : \tau$, *and* $\mathcal{N}(g)_\phi = (G, \_)$, *then for all* $(n \longrightarrow n_\alpha\ \overline{n}) \in G$, *we have* $\alpha \in \mathsf{fv}(g)$.

From this lemma, we can derive that productions from normalizing any closed g is of the desired form (i.e., $t\ \overline{n}$ or ϵ). We can further prove that there is at most one ϵ production for every nonterminal, and productions of each nonterminal start with distinct terminals. The proof relies on the following two lemmas, showing that normalizing offers the same guarantees as the typing-based approach.

Lemma 3.2. *Given* $\Gamma; \Delta \vdash g : \tau$, *and* $\mathcal{N}(g)_\phi = (G,\ n)$, *if* $n \xrightarrow{G}{}^{*} \epsilon$, *then* $\tau.\textsc{Null} = \mathsf{true}$.

Lemma 3.3. *If* $\Gamma;\Delta \vdash g : \tau$, *and* $\mathcal{N}(g) = (G,\ n)$, *and* $(n \to t\,\overline{n}) \in G$, *then* $t \in \tau.\textsc{First}$.

With that, we conclude that normalizing produces deterministic GNF.

Theorem 3.4. *If* $\Gamma;\Delta \vdash g : \tau$, *and* $\mathcal{N}(g)_\phi = (G, \_)$, *then* G *is of deterministic GNF.*

3.4.2 Totality.

We next prove that normalization is a total function on typed context-free expressions.

Theorem 3.5. *If* $\Gamma;\Delta \vdash g : \tau$, *and* $\mathrm{dom}\,(\Gamma,\Delta) \subseteq \mathrm{dom}\,(\phi)$, *then there exist* G *and* n *such that* $\mathcal{N}(g)_\phi = (G,\ n)$.

This lemma may seem trivial, but notice that G imposes syntactic restrictions on the form of productions. For example, normalizing $g_1 \cdot g_2$ produces $N_i\,n_2$, which, in order for itself to be a valid norm form, must have N_i not to be ϵ. This is where types come into play: well-typedness of $g_1 \cdot g_2$ guarantees that $\textsc{Null}$ of g_1 is false, which in turns establishes that N_i cannot be ϵ.

3.4.3 Correctness.

Finally, we prove that normalization preserves the semantics of well-typed context-free expressions. In particular, Krishnaswami and Yallop defined the denotational semantics of context-free expressions as a language $[\![g]\!]_\gamma$, where γ gives interpretation of free variables in g. In the following theorem, w denotes a word, and the notion $n \xrightarrow{G}{}^{*}\ w$ defines a derivation from the start symbol n following the grammar G.

Theorem 3.6. *If* $\bullet;\bullet \vdash g : \tau$, *and* $\mathcal{N}(g)_\bullet = (G,\ n)$, *then for all* w, *we have* $w \in [\![g]\!]_\bullet$ *if and only if and* $n \xrightarrow{G}{}^{*}\ w$.

The proof is done on a generalized form of the theorem, with non-empty type contexts and their corresponding interpretations of variables, and a non-empty environment ϕ with extra grammars for free metavariables. The key challenge in the proof is to relate precisely the interpretations of variables to the grammar rules for metavariables. The proof is then structured by first induction on the length of w, and then induction on g. We refer interested readers to the appendix for more details.

4 Fusion

We now turn to lexer-parser fusion, our central contribution. This section shows how to start with a separately-defined lexer and parser, connected together via tokens, and produce entirely token-free code, in which the only branches involve inspecting individual characters.

Fusion acts on a lexer and a normalized parser, and produces a grammar representation in which regular expressions take the place of tokens. As an example, Figure 8 shows the result of fusing the s-expression lexer of Section 3 with the normalized s-expression grammar of Section 3. Three points deserve particular attention. First, the tokens ATOM, LPAR and RPAR in the grammar have been replaced with the

$$
\begin{array}{rcl}
\text{sexp} & ::= & \texttt{id} \\
& | & \texttt{space sexp} \\
& | & \texttt{(sexps rpar} \\
\text{rpar} & ::= & \texttt{space rpar} \\
& | & \texttt{)} \\
\text{sexps} & ::= & \texttt{id sexps} \\
& | & \texttt{space sexps} \\
& | & \texttt{(sexps rpar sexps} \\
& | & \texttt{?¬(id | space | ()}
\end{array}
$$

Figure 8. Fused s-expression lexer & grammar

$$
\begin{array}{lcl}
\text{lexers} & \mathrm{L} & ::= \quad \overline{r \Rightarrow t},\ \textbf{skip}\ r \\
\text{fused form} & \mathrm{f} & ::= \quad r\overline{n} \ \mid ?r \\
\text{fused grammar} & \mathrm{F} & ::= \quad \overline{n_i \to f_i}^{\,i}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{F}(\overline{n \to t\overline{n}} \cup n \to \epsilon, n)_L & = & \overline{n \to r\overline{n}} \\
& & \cup\ n \to r_s n \\
& & \cup\ n \to ?\hat{r} \\
& & \text{where} \quad \overline{r \Rightarrow t},\ \textbf{skip}\ r_s \in L \\
& & \qquad \hat{r} = \neg((\bigvee \overline{r}) \vee r_s)
\end{array}
$$

Figure 9. Lexer-parser fusion

regular expressions `id`, `(` and `)` associated with those tokens in the lexer. Second, for each non-terminal A there is an extra production $A ::= \texttt{space}\ A$, corresponding to the **Skip** rule in the lexer. Finally, the ϵ rule for sexps has been replaced with the complement of the three regular expressions that appear at the start of the right hand side of the other productions for sexps.

4.1 The fusion algorithn

Figure 9 formally defines the fusion algorithm.

We assume a more restrictive definition of lexers than the interface in Section 2 provides. In particular, we assume that rules are disjoint on the left (i.e. there is no string that is matched by more than one regular expression in a set of rules), and disjoint on the right (i.e. there is exactly one **Skip** rule, and no token appears in more than one **Return** rule). It is easy to transform a lexer that does not obey these constraints into an equivalent lexer that does, so there is no need to restrict the interface exposed to the user.

With the lexer thus canonicalized and the parser translated into deterministic GNF, it is straightforward to define fusion.

A fused grammar maps nonterminals to fused forms, where a fused form is either a regex followed by a sequence of nonterminals, or a lookahead production formed from a regex. The fusing function $\mathcal{F}(G,n)_L$ then operates on the productions G of a single non-terminal n, replacing each production $n \to t\overline{n}$ with a new production $n \to r\overline{n}$ (retrieving the regex r that is associated with the token t in the lexer L), adding an

additional production $n \longrightarrow r_s n$ for the **skip** regex r_s (which may be $\bot$), and adding a lookahead production $n \longrightarrow ?\hat{r}$ for the regex $\hat{r}$ that is the complement of all the regexes that appear in other productions for n.

4.2 Specializing the lexer

The fusion function implicitly specializes the lexer to each non-terminal in the normalized grammar. Lexing rules that return tokens do not appear in productions for a non-terminal n are discarded. For example, in the s-expression grammar, the *rpar* nonterminal has only a single production, which begins with the terminal RPAR, and so the lexing rules that return other non-terminals are discarded:

```
  id  ⇒  Return ATOM
space ⇒  Skip
   (  ⇒  Return LPAR
   )  ⇒  Return RPAR
```

However, the **Skip** rule is retained, since skipped characters can precede any token.

Canonicalizing the lexer to enforce disjointness simplifies this discarding of rules. The discarded regexes are effectively incorporated into the lookahead regex $\hat{r}$, if there is a lookahead (ϵ) rule associated with the nonterminal.

4.3 Parsing with fused grammars

Parsing with fused grammars in `flap` requires keeping track of the state that arises in both a lexer and a parser. The parser state includes a current nonterminal and a stack of nonterminals that serve as continuation after the current nonterminal is matched. The lexer state includes a vector of regex derivatives (Section 2.1), which are used to simultaneously match strings against the set of tokens to which the leftmost part of a production for the current nonterminal might expand.

Altogether, the parsing function takes five inputs: the grammar, the productions for the current nonterminal (possibly updated with the vector of derivatives), the stack of nonterminals, the beginning of the current token (used during lookahead), and the input string.

Here is an excerpt of the parsing algorithm, for the case where both the stack and the input string are empty:

$$
\begin{aligned}
&\mathcal{P}(G, \overline{n \longrightarrow r\overline{n}} \mid ?\,\hat{r}, [], t, []) \quad = \\
&\quad \text{if } \nu(\hat{r}) \text{ then } \textsc{true} \\
&\quad \text{else if } \exists r\cdot \in \overline{r\overline{n}}\ s.t.\ \nu(r) \text{ then } \textsc{true} \\
&\quad \text{else if } \nexists r\overline{n} \in \overline{r\overline{n}}\ s.t.\ \nu(r) \text{ then } \textsc{false} \\
&\quad \text{else } \mathcal{P}(G, \overline{n' \longrightarrow r'\overline{n'}}, \hat{r}', \overline{n}, [], []) \\
&\quad \text{where } G.n' = \overline{n' \longrightarrow r'\overline{n'}} \mid ?\,\hat{r}' \\
&\quad\ rn'\overline{n} \in \overline{r\overline{n}} \ \wedge \nu(r)
\end{aligned}
$$

Whether parsing is successful depends on the nullability of various regexes involved. If the lookahead regex $\hat{r}$ is nullable, or if there is some production $n \longrightarrow r$ and r is nullable, then parsing succeeds. Alternatively, if there is no producuction involving a nullable regex for the current nonterminal, parsing fails. Finally, if there is a production $n \longrightarrow rn'\overline{n'}$, and r is nullable, then parsing continues with the new nonterminal n', leaving $\overline{n'}$ on the stack.

Here is a second excerpt, for a case where the input is not empty.

$$
\begin{aligned}
&\mathcal{P}(G, \overline{n \longrightarrow r\overline{n}} \mid ?\,\hat{r}, S, t, c :: cs) \quad = \\
&\quad \text{if } \exists r'\overline{n'} \in \overline{\partial_c\ r\overline{n}}\ s.t.\ r \neq \bot \\
&\quad \text{then } \mathcal{P}(G, \overline{n \longrightarrow \partial_c\ r\overline{n}} \mid ?\,\partial_c\ \hat{r}, S, t, cs) \\
&\quad \text{else } \ldots
\end{aligned}
$$

In this excerpt, if any of the derivatives of the terminals in the productions of the current nonterminal n is non-empty, parsing consumes the token and transitions to a new state. In this new state, the productions of n are modified to be the derivative of the previous productions of n. In this way, parsing proceeds character-by-character, leaving the parsing state otherwise unchanged.

4.4 Staging

Finally, `flap` uses MetaOCaml's staging facilities [14] to generate code for the fused grammar. Krishnaswami and Yallop applied a number of binding-time improvements such as CPS conversion [2, 19] and The Trick [6] to improve the results of staging. These techniques are not needed in `flap`, whose normalized grammar representation is already amenable to generating optimized code. Furthermore, `flap` does not rely on compiler optimizations to further simplify the code it generates; instead, it directly generates efficient code, containing no indirect calls, no higher-order functions and no allocation, except where these elements are inserted by the user of `flap` in semantic actions.

The staging step in `flap` generates one function for each parser state (i.e. for each pair of a nonterminal and a regex vector), following the algorithm in Section 4.3, but eliminating information that is statically known, such as the nullability and derivatives of the regexes associated with each state. Here is an example, from the generated code for the s-expression vector:

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

Each generated function follows a similar pattern: it checks for the end of input, and (if input remains), matches the first character and either transitions to another parser state or terminates (successfully or otherwise). The four arguments to the parser represent the beginning of the current token (to support backtracking in the lookahead transition), the current index, the input length, and the input itself.

4.4.1 Incorporeal tokens.

Although tokens themselves do not appear in the generated code, the payload associated with each token is made available to semantic actions. For each token matched by the lexer, a quoted expression ≪ `String.sub s i j` ≫ (with suitable bounds `i` and `j`) denoting the matched substring of the input `s` is made available to semantic actions to be incorporated into the generated code (e.g. for the ATOM token) or discarded (e.g. for the LPAR token) as appropriate.

4.4.2 Generating mutual recursion.

One limitation of multi-stage languages based on quotation, such as MetaOCaml, is the lack of facility for generating code that is not based on expression nesting. The binding group that `flap` generates is a typical example of this type of code: a group of n mutually-recursive bindings is an indivisible unit that does not have a smaller group of bindings as a sub-expression.

The solution in `flap` is to use *letrec insertion* [32], via an indexed fixed point operator `letrec` with the following type:

```
type resolver = ∀α.α index → α code
val letrec : (resolver → ∀γ.γ index → γ code)
             → (resolver → β code) →β code
```

Here `resolver` is the type of a function that maps typed *indexes* to typed `code`; the `index` type is a parameter to the letrec insertion library.

The `letrec` function itself takes two arguments, which respectively generate the bindings and body of a **`let rec`** expression:

```
let rec (* bindings *)
        b1 = e1 and b2 = e2 and ... and bn = en
        (* body *)
        in e
```

In each case, `letrec` makes a `resolver` available to the generating function, so that the generation of each binding can generate additional bindings. Each call to a resolver generates a new definition (if the index has not been previously seen) for the set of bindings under construction, and returns the generated variable to the caller.

For `flap`, indexes correspond to the statically-known components of parser states: each index carries a non-terminal and a vector of regexes; invoking the resolver with such an index generates the code for the corresponding function body. The overall effect is similar to the generation of the staged regex matcher described by Rompf et al. [23].

4.4.3 Optimizing the end-of-input check.

In OCaml, strings are represented as structured values; they carry several pieces of metadata, including their length. However, the runtime additionally ensures that OCaml strings are null-terminated, to ease interoperability with C. The null terminator facilitates an alternative scheme for code generation: rather than checking the length at the beginning of each fucntion as shown above, the code generated by `flap` can check for the null character alongside other alternatives. The code can then check the string length only when the null character is encountered (to distinguish the case where the string contains a null character other than the terminator):

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

As the experiments reported in Section 5 show, in some circumstances this alternative approach has significantly better performance.

5 Evaluation

This section describes experiments to investigate the performance of `flap`, and shows that lexer-parser fusion drastically improves performance. Many parser combinator libraries suffer from very poor performance, but the experiments described here show that combinator parsing does not need to be slow.

Three key techniques account for `flap`'s speed. First, Krishnaswami and Yallop's type system ensures that the time taken for parsing is linear in the length of the input, a substantial advantage over libraries that require backtracking. Second, staging eliminates the overhead arising from parsing abstractions, generating parsing code that is specialized to a particular grammar. Finally, lexer-parser fusion eliminates the overhead arising from defining lexers and parsers separately: in particular, it avoids the materialization of tokens, and eliminates all branching except for approximately one branch on each character in the input.

Krishnaswami and Yallop showed that the first two of these techniques can be used to build a parser combinator library that outperforms code generated by `ocamlyacc`. We focus here on the question of how much additional performance benefit arises from lexer-parser fusion.

5.1 Benchmarks

We build on the benchmark suite published by Krishnaswami and Yallop [17], adding implementations of each benchmark for `flap`, and extending the suite with an additional benchmark for parsing CSV files.

We make no comparison with an implementation of unstaged parser combinators, which Krishnaswami and Yallop found to be significantly slower than their staged implementation, and between 4.5 and 125 times slower than ocamlyacc.

Instead, for each benchmark we compare four implementations:

(a) the staged parser implementation from [17], in which parsers are built from two applications of the typed combinators, to build a scanner and a parser
(b) an implementation generated by `ocamllex` and `ocamlyacc`

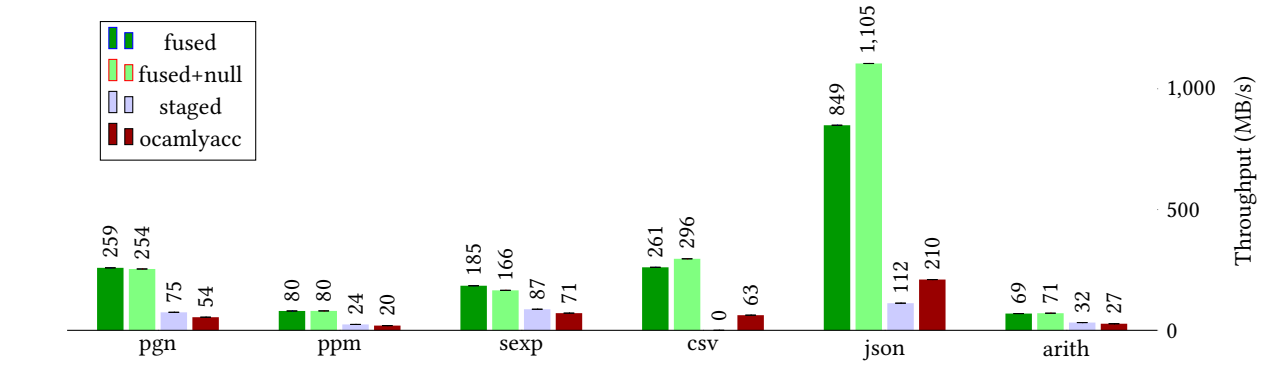


Figure 10. Parser throughput: staged combinators, fused combinators and `ocamlyacc`/`ocamllex`.

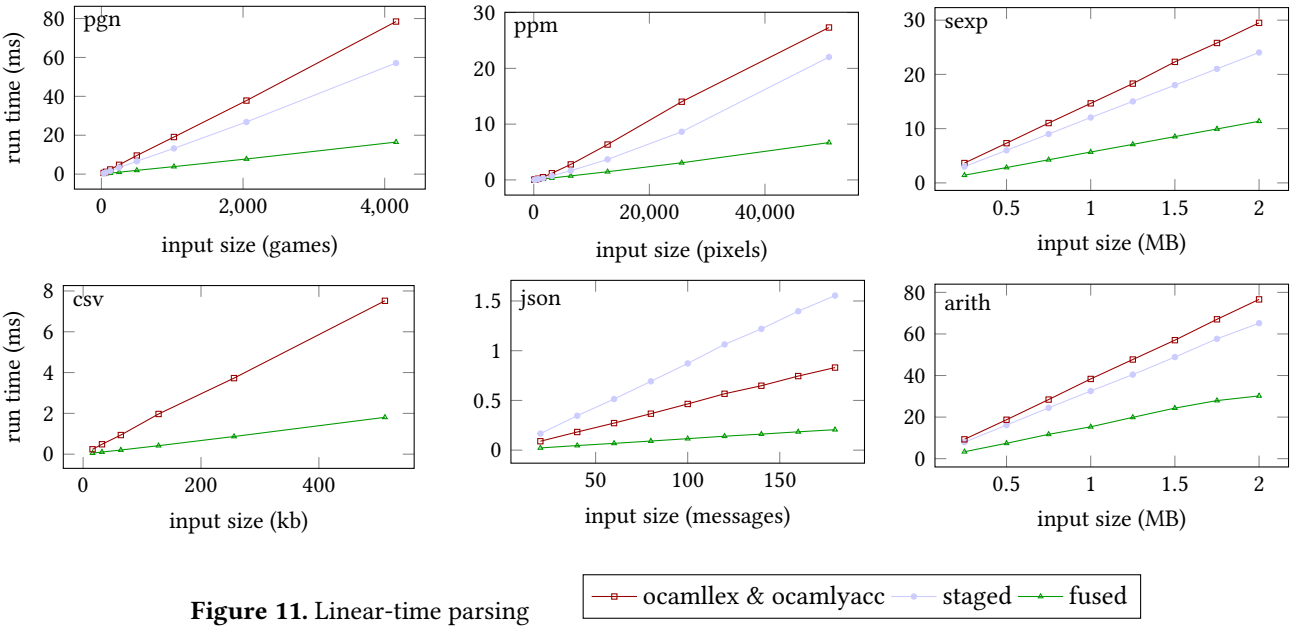


Figure 11. Linear-time parsing

(c) an implementation created by `flap`
(d) an implementation created by `flap`, with the additional optimization described in Section 4.4.3.

Since the final three implementations all use the standard parser combinator interface (Figure 4), the structure of the grammar is identical across implementations.

The benchmarks are as follows:

1. (pgn) Chess game descriptions in Portable Game Notation format. The semantic actions extract the result of each game. The input is a corpus of 6759 Grand Master games.
2. (ppm) Image files in Netpbm format. The semantic actions validate the non-syntactic constraints of the format, such as colour range and pixel count.
3. (sexp) S-expressions with alphanumeric atoms. The semantic actions of the parser count the number of atoms in each s-expression.
4. (csv) A parser for the Comma-Separated Value format. The grammar conforms quite closely to RFC 4180 [25], but makes the terminating CRLF mandatory and does not treat headers specially. The semantic actions check that each row contains the same number of fields. There is no implementation using Krishnaswami and Yallop's combinators for this benchmark, because more than a single character of lookahead is needed to distinguish escaped (i.e. repeated) double-quotes "" from unescaped quotes ", and so the lexer cannot be implemented with typed context-free expressions without substantial changes to its structure. The lexer interface used in `flap` (Section 2.2) does not suffer from this limitation.
5. (json) A parser for JavaScript Object Notation (JSON). The semantic actions count the number of objects represented in the input. Following Krishnaswami

and Yallop, we use the simple JSON grammar given by Jonnalagedda et al. [13].

6. (arith) A miniature programming language with constructs for arithmetic, comparison, let-binding and branching. The semantic actions evaluate the parsed expression.

5.2 Running time

Figure 10 shows the throughput of the four implementations using the six benchmark grammars. For the benchmarks that are taken from [17] we use the test corpora from the same source. For the CSV benchmark we have generated a set of files of various sizes and dimensions, using a random variety of textual and numeric data.

The benchmarks were compiled with BER MetaOCaml N111 and run on an Intel i7-10700K machine with 16GB memory running Debian Linux, using the `Core_bench` micro-benchmarking library [12].

As the graph shows, our experiments confirm the results reported by Krishnaswami and Yallop: the staged implementation of typed context-free expressions generally outperforms `ocamlyacc`. The addition of lexer-parser fusion makes `flap` considerably faster than both typed CFEs and `ocamlyacc`, reaching over 1GB/s (a little under 3.5 cycles per byte) on the `json` benchmark when the end-of-input optimization (Section 4.4.3) is included.

The precise ratio between the performance of the implementations varies across architectures and benchmarks. Table 1 table summarises the performance improvement of `flap` vs `ocamlyacc` across three machines:

	Intel i7-10700K	AMD FX 8320	Arm M1
pgn	4.8	5.6	5.8
ppm	4.1	5.1	4.6
sexp	2.6	4.0	4.0
csv	4.2	4.8	6.2
json	4.0	3.8	8.9
arith	2.5	2.8	3.2

Table 1. `flap` speedup vs `ocamlyacc`

Linear-time parsing. Finally, as Figure 11 illustrates, due to Krishnaswami and Yallop's type system, parsers built with `flap` are guaranteed to execute in time linear in the length of their input.

5.3 Code size

Running time is not the only important measure of usefulness for parsing. A second requirement is code size: if parsing tools are to be usable in practice, it is essential that they do not generate unreasonably large code.

There are several reasons to be apprehensive about the size of code generated by `flap`. First, conversion to Greibach Normal Form is well known to substantially increase the size of grammars; for example, in the procedure given by Blum and Koch [1] the result of converting a grammar G has size $O(|G|^3)$. Second, the fusion process is inherently duplicative, repeatedly copying the lexer rules into the various grammar productions. Finally, experience in the multi-stage programming community shows that it is easy to inadvertently generate extremely large programs, since antiquotation makes it easy to duplicate terms.

However, measurements largely dispel these concerns. Table 2 gives the size of the representations of the benchmark parsers at various stages in `flap`'s pipeline. The leftmost pair of columns of figures shows the size of the input parsers, measured as the number of lexer rules (including both **Return** and **Skip** rules) and the number of context-free expression nodes, as described in Section 2. The pair of columns to the right shows the number of nonterminals and productions after the grammar is converted to Deterministic GNF using the procedure in Section 3. As the figures show, the normalization algorithm for typed context-free expressions does not produce the drastic increases in size that occurs in the more general form of conversion to GNF. The next column to the right shows the size of the grammar after fusion (Section 4). Fusion does not alter the number of nonterminals, but it can add productions: for example, the **Skip** rules in the s-expression 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 context-free expressions in the input reveals a fairly unalarming relationship: with one exception (pgn), the ratio between the two barely exceeds 2.

Sharing. The entries for *pgn* and *arith* hint at opportunities for further improvement. In both cases, the number of context-free expressions that make up the grammar (95 and 143) is surprisingly high, since both languages are fairly simple. Inspecting the implementations of the grammars reveals the cause: in several places, the combinators that construct the grammar duplicate subexpressions. For example, here is the implementation of a Kleene plus operator used in *pgn*:

```
let oneormore e = (e >>> star e) ...
```

Normalization turns these two occurrences of `e` into multiple entries in the normalized form, and ultimately to multiple functions in the generated code.

The core problem is that the parser combinator interface (Section 2) does not provide a way to express sharing of subgrammars. Since duplication of this sort is common, it is likely that extending `flap` with facilities to express and maintain sharing could substantially reduce the size of generated code.

	Input		Normalized		Fused	Output
Benchmark	Lexer rules	Context-free exprs	Nonterminals	Productions	Productions	Generated 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 2. Sizes of inputs, intermediate representations, and generated code

6 Related work

6.1 Deterministic Greibach Normal Form

There are several longstanding results related to deterministic variants of Greibach Normal Form. For example, Geller et al. [10] show that every strict deterministic language can be given a strict deterministic grammar in Greibach Normal Form, and Nijholt [20] gives a translation into Greibach Normal Form that preserves strict deterministicness. The distinctive contributions of this paper are the new normal form that is well suited to fusion, and the compositional normalization procedure from typed context-free expressions, allowing deterministic GNF to be used in the implementation of parser combinators.

6.2 Combining lexers and parsers

The work most closely related to ours, by Casinghino and Roux [4] investigates the application of stream fusion techniques to parser combinators. Like the current work, their investigation is inspired by typed context-free expressions, and they report speed improvements over the results reported by Krishnaswami and Yallop [17] (29% faster for s-expressions and 9% faster for JSON). A significant difference between their work and ours is that they approach fusion as a traditional optimization problem, in which transformations are applied to code that satisfies certain heuristics, and are not applied in more complex cases. In contrast, we treat lexer-parser fusion as a sequence of total transformations that is guaranteed to convert every input (i.e. every parser) into a form that enjoys pleasant performance properties.

Another line of work, on *Scannerless GLR parsing* [8, 28], also aims to eliminate the boundary between lexers and parsers, both in the interface and the implementation. The principal aim is to provide a principled way to handle lexical ambiguity, in contrast to our focus on performance.

Context-aware scanning, introduced by Van Wyk and Schwerdfeger [29] is another variant on the parser-scanner interface focused on disambiguation; it passes contextual information from the parser to the scanner about the set of valid tokens at a particular point, in a similar way to the lexer specialization in Section 4.2 of this paper. However, Van Wyk and Schwerdfeger's framework goes further, and allows the automatic selection of a lexer (not just a subset of lexing rules) based on the parsing context.

6.3 Fusion

The notion of fusion, in the sense of merging computations to eliminate intermediate structures, has been applied in several domains, including query engines [26], GPU kernels [9] and tree traversals [24].

Perhaps the most widespread is stream fusion, which appears to have originated with Wadler's deforestation [31], and has since been successfully applied as both a traditional compiler optimization [5] and as a staged library [15] that provides guarantees similar to those we give here for parsers.

6.4 Parser optimization

Finally, in contrast to the constant-time speedups resulting from lexer-parser fusion, we note an intriguing piece of work by Klyuchnikov [16] that applies two-level-supercompilation to parser optimization, leading to asymptotic improvements.

7 Future work

There are a number of promising avenues for future work.

First, extending `flap`'s rather minimal lexer and parser interfaces to support common needs such as left-recursive grammars, lexers and parsers with multiple entry points, mechanisms for maintaining state during parsing, and more expressive lexer semantic action could make the library substantially more usable in practice.

Second, building on the proofs of normalization correctness in Section 3 to cover the whole of `flap`, we plan to formally establish that the code generated by Section 4 faithfully represents the semantics of the combinators in Section 2.

Third, applying the ideas in this paper to more powerful parsing algorithms such as LR(1) and LALR(1), and incorporating them into traditional standalone parser generator (rather than a staged library) could make lexer-parser fusion available to many more software developers.

Finally, it may be that fusion can be extended to longer pipelines than the lexer-parser interface that we investigate here. Might it be possible to fuse together (for example) decompression, unicode decoding, lexing and parsing into a single computation?

References

[1] Norbert Blum and Robert Koch. 1999. Greibach Normal Form Transformation Revisited. *Information and Computation* 150, 1 (1999), 112–118. https://doi.org/10.1006/inco.1998.2772

[2] Anders Bondorf. 1992. Improving Binding Times Without Explicit CPS-conversion. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming* (San Francisco, California, USA) *(LFP '92)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/141471.141483

[3] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *J. ACM* 11, 4 (Oct. 1964), 481–494. https://doi.org/10.1145/321239.321249

[4] Chris Casinghino and Cody Roux. 2020. ParTS: Final Report. HR001120C0016 - Final Report.

[5] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream fusion: from lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, Ralf Hinze and Norman Ramsey (Eds.). ACM, 315–326. https://doi.org/10.1145/1291151.1291199

[6] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. 1996. Eta-Expansion Does The Trick. *ACM Trans. Program. Lang. Syst.* 18, 6 (1996), 730–751. https://doi.org/10.1145/236114.236119

[7] Rowan Davies and Frank Pfenning. 1996. A Modal Analysis of Staged Computation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 258–270. https://doi.org/10.1145/237721.237788

[8] Giorgios Economopoulos, Paul Klint, and Jurgen J. Vinju. 2009. Faster Scannerless GLR Parsing. In *Compiler Construction, 18th International Conference, CC 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5501)*, Oege de Moor and Michael I. Schwartzbach (Eds.). Springer, 126–141. https://doi.org/10.1007/978-3-642-00722-4_10

[9] Jiri Filipovic, Matus Madzin, Jan Fousek, and Ludek Matyska. 2015. Optimizing CUDA code by kernel fusion: application on BLAS. *J. Supercomput.* 71, 10 (2015), 3934–3957. https://doi.org/10.1007/s11227-015-1483-z

[10] Matthew M. Geller, Michael A. Harrison, and Ivan M. Havel. 1976. Normal forms of deterministic grammars. *Discret. Math.* 16, 4 (1976), 313–321. https://doi.org/10.1016/S0012-365X(76)80004-0

[11] Sheila A. Greibach. 1965. A New Normal-Form Theorem for Context-Free Phrase Structure Grammars. *J. ACM* 12, 1 (Jan. 1965), 42–52. https://doi.org/10.1145/321250.321254

[12] Christopher S. Hardin and Roshan P. James. 2013. Core_bench: Micro-Benchmarking for OCaml. OCaml Workshop.

[13] Manohar Jonnalagedda, Thierry Coppey, Sandro Stucki, Tiark Rompf, and Martin Odersky. 2014. Staged Parser Combinators for Efficient Data Processing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 637–653. https://doi.org/10.1145/2660193.2660241

[14] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml - System Description. In *FLOPS 2014 (LNCS, Vol. 8475)*, Michael Codish and Eijiro Sumii (Eds.). Springer, 86–102. https://doi.org/10.1007/978-3-319-07151-0_6

[15] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 285–299. https://doi.org/10.1145/3009837

[16] Ilya Klyuchnikov. 2010. Towards effective two-level supercompilation. Preprint 81. Keldysh Institute of Applied Mathematics, Moscow.

[17] Neelakantan R. Krishnaswami and Jeremy Yallop. 2019. A typed, algebraic approach to parsing, See [18], 379–393. https://doi.org/10.1145/3314221.3314625

[18] Kathryn S. McKinley and Kathleen Fisher (Eds.). 2019. *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM. https://doi.org/10.1145/3314221

[19] Kristian Nielsen and Morten Heine Sørensen. 1995. Call-By-Name CPS-Translation As a Binding-Time Improvement. In *Proceedings of the Second International Symposium on Static Analysis (SAS '95)*. Springer-Verlag, London, UK, UK, 296–313. http://dl.acm.org/citation.cfm?id=647163.717677

[20] Anton Nijholt. 1979. Strict Deterministic Grammars and Greibach Normal Form. *J. Inf. Process. Cybern.* 15, 8/9 (1979), 395–401.

[21] Scott Owens, John H. Reppy, and Aaron Turon. 2009. Regular-expression derivatives re-examined. *J. Funct. Program.* 19, 2 (2009), 173–190. https://doi.org/10.1017/S0956796808007090

[22] Alban Reynaud, Gabriel Scherer, and Jeremy Yallop. 2021. A practical mode system for recursive definitions. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434326

[23] Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. 2013. Optimizing data structures in high-level programs: new directions for extensible compilers based on staging. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 497–510. https://doi.org/10.1145/2429069.2429128

[24] Laith Sakka, Kirshanthan Sundararajah, Ryan R. Newton, and Milind Kulkarni. 2019. Sound, fine-grained traversal fusion for heterogeneous trees, See [18], 830–844. https://doi.org/10.1145/3314221.3314626

[25] Yakov Shafranovich. 2005. Common Format and MIME Type for Comma-Separated Values (CSV) Files. RFC 4180. https://doi.org/10.17487/RFC4180

[26] Amir Shaikhha, Mohammad Dashti, and Christoph Koch. 2018. Push versus pull-based loop fusion in query engines. *J. Funct. Program.* 28 (2018), e10. https://doi.org/10.1017/S0956796818000102

[27] Walid Taha. 1999. *Multi-Stage Programming: Its Theory and Applications*. Technical Report.

[28] Mark van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. 2002. Disambiguation Filters for Scannerless Generalized LR Parsers. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2304)*, R. Nigel Horspool (Ed.). Springer, 143–158. https://doi.org/10.1007/3-540-45937-5_12

[29] Eric R. Van Wyk and August C. Schwerdfeger. 2007. Context-aware Scanning for Parsing Extensible Languages. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering* (Salzburg, Austria) *(GPCE '07)*. ACM, New York, NY, USA, 63–72. https://doi.org/10.1145/1289971.1289983

[30] Philip Wadler. 1985. How to Replace Failure by a List of Successes. In *Proc. of a Conference on Functional Programming Languages and Computer Architecture* (Nancy, France). Springer-Verlag, Berlin, Heidelberg, 113–128.

[31] Philip Wadler. 1990. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science* 73, 2 (1990), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A

[32] Jeremy Yallop and Oleg Kiselyov. 2019. Generating Mutually Recursive Definitions. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Cascais, Portugal) *(PEPM 2019)*. ACM, New York, NY, USA, 75–81. https://doi.org/10.1145/3294032.3294078