`flap`: A Deterministic Parser with Fused Lexing

JEREMY YALLOP, University of Cambridge, UK
NINGNING XIE, University of Toronto, Canada
NEEL KRISHNASWAMI, University of Cambridge, UK

Lexers and parsers are typically defined separately and connected by a token stream. This separate definition is important for modularity and reduces the potential for parsing ambiguity. However, materializing tokens as data structures and case-switching on tokens comes with a cost.

We show how to *fuse* separately-defined lexers and parsers, drastically improving performance without compromising modularity or increasing ambiguity. We propose a deterministic variant of Greibach Normal Form that ensures deterministic parsing with a single token of lookahead and makes fusion strikingly simple, and prove that normalizing context free expressions into the deterministic normal form is semantics-preserving. Our staged parser combinator library, `flap`, provides a standard interface, but generates specialized token-free code that runs two to six times faster than `ocamlyacc` on a range of benchmarks.

CCS Concepts: • **Software and its engineering → Parsers**; **Software performance**; • **Theory of computation → Parsing**.

Additional Key Words and Phrases: parsing, lexing, multi-stage programming, optimization, fusion

ACM Reference Format:
Jeremy Yallop, Ningning Xie, and Neel Krishnaswami. 2023. `flap`: A Deterministic Parser with Fused Lexing. *Proc. ACM Program. Lang.* 7, PLDI, Article 155 (June 2023), 24 pages. https://doi.org/10.1145/3591269

1 INTRODUCTION

Software systems are easiest to understand when their components have clear interfaces that hide internal details. For example, a typical compiler uses a separate lexer and parser to reduce parsing ambiguity [Aho et al. 2007], and connects the two components via a token stream.

Unfortunately, hiding internal details can also reduce optimization opportunities. For parsers, 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 and typically need only one token at any time. However, even in this case, materializing tokens as data structures and case-switching on tokens comes with a cost.

In this paper, we present the following contributions:

- We present a transformation that significantly improves parsing performance by fusing together a separately-defined lexer and a parser, entirely eliminating tokens.
  (1) We propose *DGNF*, a *Deterministic* variant of *Greibach Normal Form* [Greibach 1965] that ensures deterministic parsing with a single token of lookahead, allowing tokens to be discarded immediately after inspection (§2.5).

Authors' addresses: Jeremy Yallop, University of Cambridge, UK, jeremy.yallop@cl.cam.ac.uk; Ningning Xie, University of Toronto, Canada, ningningxie@cs.toronto.edu; Neel Krishnaswami, University of Cambridge, UK, Neel.Krishnaswami@cl.cam.ac.uk.

2475-1421/2023/6-ART155
https://doi.org/10.1145/3591269

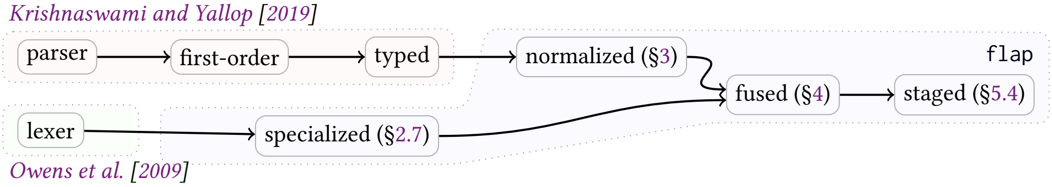


Fig. 1. Architecture of flap

(2) We formalize a *normalization* process that elaborates context-free expressions into DGNF, and prove that the elaboration is well-defined and preserves semantics (§3).
(3) We present *lexer-parser fusion*, which transforms a separately-defined lexer and *normalized* parser into a single piece of code that is specialized for calling contexts, avoids materializing tokens, and branches only on individual characters, not intermediate structures (§4).

- We implement the transformation in a parser combinator library, `flap` (***f****used* ***l****exing* ***a****nd* ***p****arsing*) (§5). The lexer and parser are built using standard tools: derivative-based lexers by Owens et al. [2009]) and typed parser combinators by Krishnaswami and Yallop [2019].
- We demonstrate the effect of our transformation: `flap` produces efficient code that runs several times faster than code produced by standard tools such as `ocamllex` and `menhir` (§6).

We survey related work in §7 and set out some directions for further development in §8.

Fig. 1 presents the novel code generation architecture of `flap`. The reader is advised to refer back to this figure while reading the rest of the article, as what it depicts will gradually come to make sense. The appendix included in the extended version of the paper [Yallop et al. 2023b] includes proofs for the lemmas in the paper.

2 OVERVIEW

2.1 Background: Parser Combinators and Typed Context-Free Expressions

Parser combinators, introduced almost four decades ago by Wadler [1985], provide an elegant way to define parsers using functions. A parser combinator library provides functions denoting token-matching, sequencing, recursion, 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. Here is a partial interface for constructing parsers (type `pa`) in this way:

```
type 'a pa
val tok: 'a tok → 'a pa                          (* token match *)
val (>>>): 'a pa → 'b pa → ('a * 'b) pa          (* sequence   *)
val fix: ('a pa → 'a pa) → 'a pa                 (* recursion *)
```

The parameterization of `pa` allows 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. However, although they enjoyed various pleasant properties (such as a rich equational theory), they suffered from potentially disastrous performance. In a recent departure from the nondeterministic tradition, Krishnaswami and Yallop [2019] define *typed context-free expressions*, whose types track properties of languages. Their design provides the standard set of parser combinators (as defined above), but adds an additional type-checking step to preclude nondeterminism and ensure linear-time parsing using a single token of lookahead.

Fig. 2 shows the typing rules from Krishnaswami and Yallop [2019], and we direct the reader to the original paper for more detailed explanations. The definition for context-free expressions (*CFE*) g is standard: $\bot$ for the empty language, ϵ for the language containing only the empty string, t for the language containing only the single-token string t, variables α, sequences $g_1 \cdot g_2$, unions $g_1 \vee g_2$,

$$
\begin{array}{rcl}
\text{Context-free expression} \quad g & ::= & \epsilon \mid t \mid \bot \mid \alpha \mid g_1 \cdot g_2 \mid g_1 \vee g_2 \mid \mu\alpha : \tau.\, g \\
\text{Types} \quad \tau & \in & \{\textsc{Null} : 2;\ \textsc{First} : \mathcal{P}(\Sigma);\ \textsc{FLast} : \mathcal{P}(\Sigma)\} \\
\text{Contexts} \quad \Gamma, \Delta & ::= & \bullet \mid \Gamma, \alpha : \tau
\end{array}
$$

$$
\begin{array}{lcl}
\tau_\epsilon & = & \{\textsc{Null} = \text{true};\ \textsc{First} = \emptyset;\ \textsc{FLast} = \emptyset\} \\
\tau_t & = & \{\textsc{Null} = \text{false};\ \textsc{First} = \{t\};\ \textsc{FLast} = \emptyset\} \\
\tau_\bot & = & \{\textsc{Null} = \text{false};\ \textsc{First} = \emptyset;\ \textsc{FLast} = \emptyset\} \\
\tau_1 \cdot \tau_2 & = & \begin{cases} \textsc{Null} & = \ \tau_1.\textsc{Null} \wedge \tau_2.\textsc{Null} \\ \textsc{First} & = \ \tau_1.\textsc{First} \ \cup \tau_1.\textsc{Null}\, ?\, \tau_2.\textsc{First} \\ \textsc{FLast} & = \ \tau_2.\textsc{FLast} \ \cup \tau_2.\textsc{Null}\, ?\, (\tau_2.\textsc{First} \cup \tau_1.\textsc{FLast}) \end{cases} \\
\tau_1 \vee \tau_2 & = & \begin{cases} \textsc{Null} & = \ \tau_1.\textsc{Null} \vee \tau_2.\textsc{Null} \\ \textsc{First} & = \ \tau_1.\textsc{First} \cup \tau_2.\textsc{First} \\ \textsc{FLast} & = \ \tau_1.\textsc{FLast} \cup \tau_2.\textsc{FLast} \end{cases} \\
\tau_1 \circledast \tau_2 & \stackrel{\text{def}}{=} & \tau_1.\textsc{FLast} \cap \tau_2.\textsc{First} = \emptyset \wedge \neg\tau_1.\textsc{Null} \\
\tau_1 \# \tau_2 & \stackrel{\text{def}}{=} & (\tau_1.\textsc{First} \cap \tau_2.\textsc{First} = \emptyset) \wedge \neg(\tau_1.\textsc{Null} \wedge \tau_2.\textsc{Null}) \\
b\, ?\, S & \stackrel{\text{def}}{=} & \text{if } b \text{ then } S \text{ else } \emptyset
\end{array}
$$

$$
\frac{}{\Gamma;\Delta \vdash \epsilon : \tau_\epsilon} \qquad \frac{}{\Gamma;\Delta \vdash t : \tau_t}
$$

$$
\frac{}{\Gamma;\Delta \vdash \bot : \tau_\bot} \qquad \frac{\alpha : \tau \in \Gamma}{\Gamma;\Delta \vdash \alpha : \tau}
$$

$$
\frac{\Gamma;\Delta, \alpha : \tau \vdash g : \tau}{\Gamma;\Delta \vdash \mu\alpha : \tau.\, g : \tau}
$$

$$
\frac{\Gamma;\Delta \vdash g_1 : \tau_1 \quad \Gamma, \Delta; \bullet \vdash g_2 : \tau_2 \quad \tau_1 \circledast \tau_2}{\Gamma;\Delta \vdash g_1 \cdot g_2 : \tau_1 \cdot \tau_2}
$$

$$
\frac{\Gamma;\Delta \vdash g_1 : \tau_1 \quad \Gamma;\Delta \vdash g_2 : \tau_2 \quad \tau_1 \# \tau_2}{\Gamma;\Delta \vdash g_1 \vee g_2 : \tau_1 \vee \tau_2}
$$

Fig. 2. Krishnaswami and Yallop's type system for context-free expressions

and the least fixed point operator $\mu\alpha : \tau.g$. A type is a triple recording NULLability, the FIRST set, and the FLAST set. Intuitively, a type is an *overapproximation* of the properties of a language. That is, a language L satisfies a type τ, if the following is true: (1) when the empty string is in L, τ.NULL is true; (2) the set of tokens that can start any string in L is a subset of τ.FIRST; (3) the set of tokens which can follow the last token of a string in L is a subset of τ.FLAST[1].

There is one typing rule for each combinator, whose types are constructed using corresponding combinators (e.g. $\tau_1 \cdot \tau_2$). The two contexts Γ and Δ restrict where variables can occur, disallowing left recursion. Specifically, when typing $\mu\alpha : \tau.\ g$, the variable α is added to Δ. But a variable α is well-typed only if $(\alpha : \tau) \in \Gamma$. The trick is that when typing $g_1 \cdot g_2$, Δ is appended to Γ, where the *separability* side condition $\tau_1 \circledast \tau_2$ ensures that g_1 cannot be empty, so that g_2 can now use α. Additionally, $\tau_1 \circledast \tau_2$ also says that τ_1.FLAST is disjoint with τ_2.FIRST, ensuring strings matched by sequenced parsers have a unique decomposition. Moreover, the side condition *apartness* $\tau_1 \# \tau_2$ on the rule for $g_1 \vee g_2$ ensures that languages matched by alternated parsers do not overlap.

2.2 Overhead of Separate Lexing and Parsing

These typing rules ensure that well-typed expressions have good asymptotic performance, supporting linear-time parsing with a single token of lookahead. However, even linear-time parsers can be inefficient, using significant resources at each parsing step. Some parsing algorithms examine state dispatch tables to determine what actions to take; similarly, Krishnaswami and Yallop's system examines the types of context-free expressions to select branches. To avoid this overhead, Krishnaswami and Yallop apply *multi-stage programming* to eliminate dispatch on type information, generating type-specialized parsing code that has performance competitive with `ocamlyacc`.

However, even with these improvements, parsing still caries considerable overhead. The main source of inefficiency is the interface between the lexer and the parser. In a typical system, a

[1] FLAST sets, originally introduced by Brüggemann-Klein and Wood [1992], are used as a alternative to the FOLLOW set traditionally used in LL(1) parsing. FLAST is *compositional*, so Krishnaswami and Yallop [2019] can calculate larger types from smaller ones. In contrast, FOLLOW is the set of tokens following a particular nonterminal, and so is a property of a grammar rather than of a language. In practice, the typed parser combinators accept languages very close to LL(1); there are some differences that only seem to show up in contrived examples.

lexer materializes each token it recognizes, then the parser branches on that token to select an action. This approach is clearly inefficient: information about which token has been recognized was available at the point that the token was created, then discarded and recovered via branching. Just *how* inefficient it is becomes apparent when we eliminate the materialization of tokens. §6 shows that `flap`'s fused lexer and parser, in which tokens are not materialized, outperforms the unfused implementation by 2 to 7 times — that is, the overhead of token materialization and associated branching accounts for the majority of parsing time.

2.3 Our Proposal: A Deterministic Parser with Fused Lexing

In this work, we take a systematic approach to fusing a lexer and a *deterministic* parser. We demonstrate our approach with `flap`, a parser combinator library that fuses

(1) a lexer based on *derivatives* of regular expressions (regexes) [Owens et al. 2009], and
(2) parser combinators for typed context free expressions (§2.1) [Krishnaswami and Yallop 2019].

Lexer-parser fusion is not inherently limited to this particular combination; it extends to other lexers for regexes and other deterministic parsers. In this paper, `flap` is restricted to LL(1) grammars, and we leave it as future work to apply `flap` to real programming languages (e.g. Python), and to adapt the fusion strategy to other grammars such as LR and other practical programming languages.

For `flap`, the characteristic properties of derivatives and typed CFEs make our implementation straightforward. First, derivatives make it straightforward to build compact deterministic automata that implement regex matchers. Specifically, the *derivative* [Brzozowski 1964] of a regex r with respect to a character c is another regex $\partial_c\ r$ that matches s exactly when r matches $c \cdot s$. Therefore, one way to construct an automaton is to take regexes r as states, with a transition from r_i to r_j via character c whenever $\partial_c\ r_i = r_j$. As Owens et al. [2009] show, lexers based on derivatives provide a practical basis for real-world lexing tools such as ml-ulex and the PLT Scheme scanner generator. We direct the reader to their work for the details about derivative-based lexers that we omit here.

Second, the types in typed context-free expressions correspond to syntactic constraints in a normal form, DGNF (§3), that serves as a basis for lexer-parser fusion. 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).

The running example. The following sections illustrate `flap`'s key ideas through a running example shown in Fig. 3. Fig. 3a presents the grammars that will be introduced and used throughout this section, with colors to help distinguish different grammars for better clarify.

2.4 Example: The Lexer and Parser for S-Expressions

Consider defining a lexer and a parser for *s-expressions* (sexps) representing tree-structured data. Sexps are either (1) atoms, or (2) a possibly-empty lists of sexps enclosed in parentheses '(' sexps ')'.

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 $r\ \&\ s$, and negation $\neg r$. A lexer L is a mapping[2] from regexes to *actions*, where an action might return a token ($r \Rightarrow \textbf{Return}\ t$), invoke the lexer recursively to skip over some input $r \Rightarrow \textbf{Skip}$, 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.

Parser. Fig. 3a repeats the definition of CFE from §2.1. Fig. 3c gives a well-typed sexp grammar that matches sequences of tokens. For simplicity, we often omit τ in $\mu\alpha : \tau.\, g$.

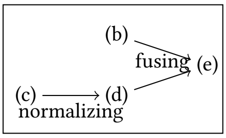
[2] We canonicalize lexers (§4) so there is no overlap between rules.

regular expression	r	::=	$\bot \mid \epsilon \mid c \mid r \cdot s \mid (r \mid s) \mid r* \mid (r \mathbin{\&} s) \mid \neg r$
lexer	L	::=	$\{ r \Rightarrow \textbf{Return } t \} \cup \{ r \Rightarrow \textbf{Skip} \}$
context-free expression	g	::=	$\bot \mid \epsilon \mid t \mid \alpha \mid g_1 \cdot g_2 \mid g_1 \vee g_2 \mid \mu\alpha : \tau . g$
DGNF grammar	D	::=	$\{ n \longrightarrow t\,\overline{n} \} \cup \{ n \longrightarrow \epsilon \}$
fused grammar	F	::=	$\{ n \longrightarrow r\,\overline{n} \} \cup \{ n \longrightarrow ?r \}$

(a) Syntax of lexers, forms, and grammars in flap

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

id $\overset{\text{def}}{=}$ `[a-z]+`
space $\overset{\text{def}}{=}$ ␣ | `\n`



(b) A s-expression lexer (2.4)

$$\mu\ \text{sexp} . (\text{LPAR} \cdot (\mu\ \text{sexps} . \epsilon \vee (\ \text{sexp} \ \cdot\ \text{sexps}\ )) \cdot \text{RPAR}) \vee \text{ATOM}$$

```
sexp  ::=  LPAR sexps RPAR   ①        sexps  ::=  sexp sexps   ③
       |   ATOM              ②               |    ϵ            ④
```

(c) A well-typed s-expression grammar (top), and its BNF form (bottom) (2.1 & 2.4)

```
sexp  ::=  LPAR sexps rpar      rpar  ::=  RPAR      sexps  ::=  LPAR sexps rpar sexps
       |   ATOM                                             |    ATOM sexps
                                                            |    ϵ
```

(d) An s-expression DGNF grammar, written in BNF form (2.5 & 2.6)

id ⇒ **Return** ATOM
space ⇒ **Skip**
(⇒ **Return** LPAR
~~) ⇒ **Return** RPAR~~

sexp ::= (sexps rpar
| id
| space sexp

~~id ⇒ **Return** ATOM~~
space ⇒ **Skip**
~~(⇒ **Return** LPAR~~
) ⇒ **Return** RPAR

rpar ::=)
| space rpar

id ⇒ **Return** ATOM
space ⇒ **Skip**
(⇒ **Return** LPAR
~~) ⇒ **Return** RPAR~~

sexps ::= (sexps rpar sexps
| id sexps
| space sexps
| ?¬(id | space | ()

(e) Fusing drops lexing rules that return non-matchable tokens (top); the fused s-expr grammar (bottom) (2.7)

Fig. 3. flap running example: s-expression lexing and parsing. Grammars are written in BNF form.

The bottom of Fig. 3c shows the BNF form of the grammar to help understanding. Intuitively, sexp stands for s-expressions, and sexps stands for lists of s-expressions. That is, sexp is either a LPAR token followed by a list of s-expressions sexps and a RPAR token, or an ATOM token; and sexps is either empty (ϵ), or a sexp followed by another list of s-expressions (sexp sexps).

The rest of this section will show how to fuse the lexer and parser. First, however, we need to present DGNF grammars.

2.5 Deterministic Parsing with DGNF

To motivate DGNF, we consider how to parse with the s-expression grammar in Fig. 3c. Linear time, one-token-lookahead, deterministic parsing requires committing to a particular branch after examining each token. However the grammar in Fig. 3c does not make it clear how to select branches by examining a single token.

For example, when parsing sexps when the first token matches LPAR, it is not immediately clear from the productions for sexps whether to pursue production ③ or production ④. Deterministic parsing systems improve this situation by analysing the grammar beforehand to calculate the branches that correspond to particular input tokens. In Krishnaswami and Yallop's case, the analysis takes the form of type inference. Each CFE is annotated with a type whose FIRST set indicates which tokens can appear at the beginning of the strings in the corresponding language. Their parsing algorithm then examines FIRST sets to select branches. Using multi-stage programming, they then improve efficiency by ensuring that FIRST sets are only examined during analysis, not during parsing itself.

In this work we take a different approach, transforming the grammar into a form in which the branch to take at each point is syntactically manifest. We call this form *Deterministic Greibach Normal Form* (DGNF), since it is a deterministic variant of GNF [Greibach 1965]. Fig. 3a shows the syntax for *DGNF grammars*. A DGNF grammar D is a set of productions that map nonterminals to normal forms, where all productions are either of form $n \longrightarrow t\ \overline{n}$ or $n \longrightarrow \epsilon$, where n is a nonterminal, t is a terminal, and $\overline{n}$ denotes $n_1 n_2 \ldots n_k (k \geq 0)$. Moreover, a DGNF grammar must also satisfy the following constraints (the formal definition of DGNF is given later in §3.2). First, for any pair of a nonterminal n and a terminal t, there is at most one production beginning $n \longrightarrow t n_1 n_2 \ldots n_k$. Second, the ϵ-production may only be used when no terminal symbol in the non-ϵ productions matches the input string.

Intuitively speaking, the constraints on the DGNF grammar are a syntactic analogue of the constraints enforced by the types in the typed CFEs. The constraints have a simple practical motivation in parsers. That is, each $n \longrightarrow t n_1 n_2 \cdots n_k$ production represents one branch that matches a distinct terminal t, and ϵ-productions represent an else branch that is taken if none of the active productive branches matches the input. With those constraints, it is evident that DGNF ensures deterministic parsing with a single token of lookahead, and branching (except for ϵ-productions) always consumes tokens immediately.

Examples. We consider a few examples. For readability, we write the grammar in BNF form, e.g. $n ::= \textsc{a}n_1n_2 \mid \textsc{b}$ corresponds to $n \longrightarrow \textsc{a}n_1n_2$ and $n \longrightarrow \textsc{b}$.

$$
(1)\ \begin{array}{lcl} n & ::= & \textsc{a}n_1n_2 \\ & \mid & \textsc{b} \\ n_1 & ::= & \textsc{c} \\ n_2 & ::= & \textsc{e} \end{array}
\quad
(2)\ \begin{array}{lcl} n & ::= & \textsc{ab}n_1 \\ n_1 & ::= & \textsc{c} \\ & & \\ & & \end{array}
\quad
(3)\ \begin{array}{lcl} n & ::= & \textsc{a}n_1 \\ & \mid & \textsc{a}n_2 \\ n_1 & ::= & \textsc{c} \\ n_2 & ::= & \textsc{e} \end{array}
\quad
(4)\ \begin{array}{lcl} n & ::= & \textsc{a}n_1n_2 \\ n_1 & ::= & \textsc{c} \\ & \mid & \epsilon \\ n_2 & ::= & \textsc{c} \end{array}
$$

Here (1) is in DGNF, while (2) (3) (4) are not. The reasons why (2) (3) are not are obvious: In (2), n starts with two terminals; in (3), n has two productions starting with A.

(4) is the most subtle case. Consider matching n with AC. First, n expands to $\textsc{a}n_1n_2$. But should n_1 then expand to C or ϵ? In a general nondeterministic grammar, it is impossible to tell simply by looking ahead at the next token C: we may first consider $n_1 \longrightarrow \textsc{c}$ and, finding that n_2 fails to match, backtrack to the other branch to consider $n_1 \longrightarrow \epsilon$ and $n_2 \longrightarrow \textsc{c}$ and succeed. However, the second constraint on DGNF grammars eliminates this choice: only $n_1 \longrightarrow \textsc{c}$ applies, and so the grammar does *not* match AC. As we will see, the formal definition of DGNF (§3.2) rules out (4) as a DGNF grammar, ensuring that parsing is deterministic. As the examples demonstrate, DGNF ensures that there is never any ambiguity about whether a production rule applies during parsing.

Fig. 3c is obviously not a DGNF grammar. So next, we discuss a normalization algorithm that normalizes a context-free expression into a DGNF grammar.

2.6 Normalizing Context-Free Expressions to DGNF Grammars

We formalize a normalization algorithm (§3) which takes a context-free expression, traverses its structure and turns it into a DGNF grammar. As an example, Fig. 3d presents the DGNF grammar of normalizing the s-expression grammar in 3c. This example illustrates several points.

First, the normalized DGNF presentation addresses the problem of repeated branching discussed in the last section. In particular, parsing sexps 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. Only in the last case is the token examined more than once: after selecting the ϵ branch that does not consume it, the token is retained until it selects a non-ϵ branch that does.

Second, while in this case it seems straightforward to check that the normalized grammar (3d) represents exactly the same language as the original context-free expression (3c), establishing correctness properties for normalization is generally difficult. A particularly challenging case is when normalizing a fixed point $\mu\alpha.\ g$. In such case, although we do not yet know the normalized grammar for α, we must proceed with normalizing g regardless. Therefore, it is necessary to "tie the knot" when the result of normalizing g becomes available, which requires us to introduce an intermediate *non-DGNF* grammar form $n \longrightarrow \alpha\ \overline{n}$, causing extra complication and subtleties during normalization. We detail the normalization process and its correctness proofs in §3.

Lastly, in what cases do we know that normalization will produce DGNF grammars? For example, it'd be difficult (if not impossible) to normalize an ambiguous grammar. Fortunately, typed context-free expressions give us enough guarantee: we prove that if a context free expression is well-typed, then the normalization will always produce a DGNF grammar. This is done by showing that DGNF indeed captures the constraints enforced by types in the typed context free expressions system.

2.7 Lexer-Parser Fusion

Now that we have the lexer, and the normalized parser, we can apply the lexer-parser fusion. Fig. 3a defines the syntax of fused grammars. The fused grammar F is a set of productions, where each production maps a nonterminal to either a regex followed by a list of nonterminals $n \longrightarrow r\ \overline{n}$, or a single-token lookahead $n \longrightarrow ?r$ that matches but not consumes tokens by r.

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. Fig. 3e fuses the s-expression lexer (3b) and the normalized parser (3d), following the steps:

(1) As the first step, fusion implicitly specializes the lexer to each nonterminal n in the normalized grammar, and lexing rules that return tokens not in productions for the nonterminal n are discarded, except for skip rules, since skipped characters can precede any token.
Example (3e top): rpar has only a single production, which begins with the terminal RPAR. We look at the lexing rules, and discard those rules that do not return RPAR, but keep the skip rule.

(2) Then, the algorithm fuses the lexing rules and the parsing rules, by substituting the tokens in the parsing rules by regexes in the lexing rules that return corresponding tokens. Moreover, skip rules generate extra productions that match an arbitrary number of skipped characters.
Example (3e bottom): the fused rpar has two branches. The first branch fuses lexing and parsing, by having the original token RPAR replaced with the regex). The second branch corresponds to the skip rule in the lexer, allowing rpar to match an arbitrary number of spaces. Observe how rpar now directly matches on characters, without referring to any tokens.

(3) For each ϵ-production, fusion generates a lookahead rule consisting of the complement of the regexes that appear at the start of the right hand side of the other productions.
Example (3e bottom): when fusing sexps, the ϵ-production $\mathrm{sexps} \longrightarrow \epsilon$ has been replaced with a lookahead rule $\mathrm{sexp} \longrightarrow ?\neg(\mathrm{id} \mid \mathrm{space} \mid ()$.

Fig. 3e presents the complete result of fusing the s-expression lexer and normalized grammar following the fusion steps described above.

Crucially, note how the representation of DGNF grammar allows fusion to be defined so concisely – it would be more difficult to fuse the original CFE (3c) with the lexer. With DGNF grammars, the constraints on the positions of terminals make it straightforward to fuse the lexing rules into the grammar without disrupting its structure. Additionally, the fused grammar inherits the properties of DGNF: the productions of a nonterminal start with distinct regexes, and an optional lookahead rule may only be used when no regexes in other productions match the input string.

2.8 Staging

In the last step, `flap` uses MetaOCaml's staging facilities to generate code for the fused grammar. Parsing with staging is very common, and various systems use some form of staging; for example, ocamlyacc computes parsing tables once in advance, not repeatedly during parsing.

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 a parsing algorithm with fused grammars, but eliminating information that is statically known, such as the nullability and derivatives of the regexes associated with each state. The DGNF representation used in `flap` also makes staging comparatively straightforward: `flap` does not involve sophisticated optimization techniques such as the binding-time improvements, 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. §5 presents the algorithm underlying `flap`'s staged parsing implementation in more detail.

3 NORMALIZING CONTEXT-FREE EXPRESSIONS

This section presents a normalization algorithm that transforms context-free expressions into DGNF grammars. The normalization sets the basis for follow-up optimizations of fusion and staging.

3.1 Normalization to DGNF

Fig. 4 presents the syntax for normal forms and the normalization algorithm.

The normal form grammar G maps nonterminals to the normal forms. Note the difference from the DGNF grammar D: G includes an extra *non-DGNF* form $n \longrightarrow \alpha\,\overline{n}$, which makes G a superset of D. As discussed in §2.6, this non-DGNF form is necessary for normalizing fixpoints, where α is interpreted as a special kind of nonterminal. As a nonterminal, α itself may appear as part of a $t\,\overline{n}$ (e.g. $t\,\alpha$). We show later that $\alpha\,\overline{n}$ is an intermediate form that is entirely eliminated in the final result, turning the grammar into DGNF. We will mostly use G in this section.

The key to normalization is the normalization function $\mathcal{N}[\![\, g \,]\!]$ that normalizes g and yields $n \Rightarrow G$, with a distinguished start nonterminal n and a normalized grammar G. There are seven cases for the seven context-free expression constructors, and each case involves allocating a fresh nonterminal (n or α) to use as the 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-expressions. Since normalization simply merges together all the production sets resulting from sub-expressions, the situation frequently

normal form	N	::=	$\epsilon \mid t\,\overline{n} \mid \alpha\,\overline{n}$
normal form grammar	G	::=	$\{\, n \longrightarrow N \,\}$
DGNF grammar	D	::=	$\{\, n \longrightarrow t\,\overline{n} \,\} \cup \{\, n \longrightarrow \epsilon \,\}$

$\mathcal{N}[\![\, g \,]\!]$ returns $n \Rightarrow G$, with a grammar G and the start nonterminal n

Each rule allocates a fresh nonterminal n, except for rule *(fix)*, which allocates a fresh α

$$
\begin{array}{lllll}
(\mathit{epsilon}) & \mathcal{N}[\![\, \epsilon \,]\!] & = & n \Rightarrow \{\, n \longrightarrow \epsilon \,\} & \\
(\mathit{token}) & \mathcal{N}[\![\, t \,]\!] & = & n \Rightarrow \{\, n \longrightarrow t \,\} & \\
(\mathit{bot}) & \mathcal{N}[\![\, \bot \,]\!] & = & n \Rightarrow \emptyset & \\
(\mathit{seq}) & \mathcal{N}[\![\, g_1 \cdot g_2 \,]\!] & = & n \Rightarrow \{\, n \longrightarrow N_1\; n_2 \mid n_1 \longrightarrow N_1 \in G_1 \,\} \cup G_1 \cup G_2 & \\
 & & & \text{where} \quad \mathcal{N}[\![\, g_1 \,]\!] = n_1 \Rightarrow G_1 \wedge \mathcal{N}[\![\, g_2 \,]\!] = n_2 \Rightarrow G_2 & \\
(\mathit{alt}) & \mathcal{N}[\![\, g_1 \vee g_2 \,]\!] & = & n \Rightarrow \{\, n \longrightarrow N_1 \mid n_1 \longrightarrow N_1 \in G_1 \,\} & \\
 & & & \quad \cup \{\, n \longrightarrow N_2 \mid n_2 \longrightarrow N_2 \in G_2 \,\} \cup G_1 \cup G_2 & \\
 & & & \text{where} \quad \mathcal{N}[\![\, g_1 \,]\!] = n_1 \Rightarrow G_1 \wedge \mathcal{N}[\![\, g_2 \,]\!] = n_2 \Rightarrow G_2 & \\
(\mathit{fix}) & \mathcal{N}[\![\, \mu\alpha.\, g \,]\!] & = & \alpha \Rightarrow \{\, \alpha \longrightarrow N \mid n \longrightarrow N \in G \,\} & \textcircled{1} \\
 & & & \qquad \cup \{\, n' \longrightarrow N\; \overline{n}' \mid n' \longrightarrow \alpha\; \overline{n}' \in G \wedge n \longrightarrow N \in G \,\} & \textcircled{2} \\
 & & & \qquad \cup\, G\backslash_{n' \to \alpha\; \overline{n}'} & \textcircled{3} \\
 & & & \text{where} \quad \mathcal{N}[\![\, g \,]\!] = n \Rightarrow G & \\
 & & & \qquad\qquad G\backslash_{n' \to \alpha\; \overline{n}'} \text{ is } G \text{ with all } n' \longrightarrow \alpha\; \overline{n}' \text{ removed for any } n' \text{ and } \overline{n}' & \\
(\mathit{var}) & \mathcal{N}[\![\, \alpha \,]\!] & = & n \Rightarrow \{\, n \longrightarrow \alpha \,\} &
\end{array}
$$

Fig. 4. Normalization of well-typed context-free expressions.

arises where some 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.

Rules *(epsilon)*, *(token)*, and *(bot)* are straightforward. For each of ϵ and t, normalization produces a grammar with a single production whose right-hand side is ϵ or t respectively. For $\bot$, normalization produces an empty grammar, with a start symbol and no productions.

Normalization of $g_1 \cdot g_2$ (rule *(seq)*) is defined compositionally in terms of the normalization of g_1 and g_2, which produces start symbols n_1 and n_2 respectively. We then want $n \longrightarrow n_1\; n_2$, i.e.:

$$(\mathit{seq1}) \quad \mathcal{N}[\![\, g_1 \cdot g_2 \,]\!] \quad = \quad n \Rightarrow \{n \longrightarrow n_1\; n_2\} \cup G_1 \cup G_2 \qquad \text{where} \quad \mathcal{N}[\![\, g_i \,]\!] = n_i \Rightarrow G_i, i = 1, 2$$

However, while this is semantically correct, $n \longrightarrow n_1\; n_2$ is not in normal form. Therefore, rule *(seq)* instead copies each production N_1 of n_1, appending to each the start symbol n_2, producing $N_1\; n_2$. Rule *(alt)* is similar, with normalization merging the productions for the start symbols n_1 of g_1 and n_2 of g_2 into the productions for the new start symbol n.

Finally, rules *(fix)* and *(var)* deal with the binding fixed point operator $\mu\alpha.\, g$ and with bound variables α. In rule *(fix)*, we assume we can always rename bound variables to avoid clashes. Normalizing $\mu\alpha.\, g$ takes place in two stages. First, the body g is normalized, yielding a start symbol n. Then, according to the semantics of fixed point, we should proceed to tie the knot by producing $\alpha \longrightarrow n$ and return α as the start symbol. That is:

$$(\mathit{fix1}) \quad \mathcal{N}[\![\, \mu\alpha.\, g \,]\!] \quad = \quad \alpha \Rightarrow \{\alpha \longrightarrow n\} \cup G \qquad \text{where} \quad \mathcal{N}[\![\, g \,]\!] = n \Rightarrow G$$

However, $\alpha \longrightarrow n$ is not in normal form so, as with *(seq)*, we instead copy the productions for n into the rules for α:

$$(\mathit{fix2}) \quad \mathcal{N}[\![\, \mu\alpha.\, g \,]\!] \quad = \quad \alpha \Rightarrow \{\alpha \longrightarrow N \mid n \longrightarrow N \in G\} \cup G \qquad \text{where} \quad \mathcal{N}[\![\, g \,]\!] = n \Rightarrow G$$

$$\dfrac{\dfrac{\dfrac{\mathcal{N}[\![\mathrm{LPAR}]\!] = \cdots \qquad \dfrac{\cdots}{\mathcal{N}[\![\mu\,\mathrm{ss}\,.\,\epsilon \vee \mathrm{s} \cdot \mathrm{ss}]\!] = \mathrm{ss} \Rightarrow \{\mathrm{ss} \longrightarrow \epsilon, \mathrm{ss} \longrightarrow \mathrm{s\,ss}\}}}{\mathcal{N}[\![\mathrm{LPAR} \cdot (\mu\,\mathrm{ss}\,.\,\epsilon \vee \mathrm{s} \cdot \mathrm{ss})]\!] = n_1 \Rightarrow \{n_1 \longrightarrow \mathrm{LPAR\,ss}, \mathrm{ss} \longrightarrow \epsilon, \mathrm{ss} \longrightarrow \mathrm{s\,ss}\}} \qquad \mathcal{N}[\![\mathrm{RPAR}]\!] = \cdots}{\mathcal{N}[\![\mathrm{LPAR} \cdot (\mu\,\mathrm{ss}\,.\,\epsilon \vee \mathrm{s} \cdot \mathrm{ss}) \cdot \mathrm{RPAR}]\!] = n_2 \Rightarrow \{n_2 \longrightarrow \mathrm{LPAR\,ss\,rpar}, \mathrm{ss} \longrightarrow \epsilon, \mathrm{ss} \longrightarrow \mathrm{s\,ss}, \mathrm{rpar} \longrightarrow \mathrm{RPAR}\}} \qquad \mathcal{N}[\![\mathrm{ATOM}]\!] = \cdots}{\dfrac{\mathcal{N}[\![(\mathrm{LPAR} \cdot (\mu\,\mathrm{ss}\,.\,\epsilon \vee \mathrm{s} \cdot \mathrm{ss}) \cdot \mathrm{RPAR}) \vee \mathrm{ATOM}]\!] = n_3 \Rightarrow \{n_3 \longrightarrow \mathrm{LPAR\,ss\,rpar}, n_3 \longrightarrow \mathrm{ATOM}, \mathrm{ss} \longrightarrow \epsilon, \mathrm{ss} \longrightarrow \mathrm{s\,ss}, \mathrm{rpar} \longrightarrow \mathrm{RPAR}\}}{\mathcal{N}[\![g]\!] = \mathrm{s} \Rightarrow \{\mathrm{s} \longrightarrow \mathrm{LPAR\,ss\,rpar}, \mathrm{s} \longrightarrow \mathrm{ATOM}, \mathrm{ss} \longrightarrow \epsilon, \mathrm{ss} \longrightarrow \mathrm{LPAR\,ss\,rpar\,ss}, \mathrm{ss} \longrightarrow \mathrm{ATOM\,ss}, \mathrm{rpar} \longrightarrow \mathrm{RPAR}\}}}$$

Fig. 5. Normalizing s-expression $g = \mu\,\mathrm{s}\,.(\mathrm{LPAR} \cdot (\mu\,\mathrm{ss}\,.\,\epsilon \vee \mathrm{s} \cdot \mathrm{ss}) \cdot \mathrm{RPAR}) \vee \mathrm{ATOM}$

But there is some extra work. In particular, productions in G might start with α (e.g. $n' \longrightarrow \alpha\,\overline{n}'$). While such form is allowed by the syntax of N, our ultimate goal is to get rid of it and turn the productions into DGNF. Now that we learn the rules of α, we can look up and substitute in G all productions that start with α. For example, if $\alpha \longrightarrow \mathrm{B}$ and $n' \longrightarrow \alpha\,\overline{n}$, then after substitution we have $n' \longrightarrow \mathrm{B}\,\overline{n}$. Note that α, as a special kind of nonterminal, may still appear in the middle of a production; for example, $n' \longrightarrow t\,\alpha$ won't get substituted. Performing the substitution would not be correct: if $\alpha \longrightarrow \mathrm{B}$, then after substitution $n' \longrightarrow t\mathrm{B}$ is not in DGNF.

Rule (*fix*) in Fig. 4 presents the final form of normalizing a fixed point. ① first copies the productions for n into the rules for α, then ② substitutes in G all productions that start with α, and ③ finally adds back all productions in G that do not start with α. As we will see, rule (*fix*) effectively guarantees that normalizing closed context-free expressions produces DGNF.

Lastly, rule (*var*) creates the singleton production $n \longrightarrow \alpha$. Combining (*fix*) with (*var*), normalization treats α as a placeholder for the productions denoted by the fixed point. Once α is known, it is replaced with its productions if necessary (as in rule (*fix*)). It is tempting to return $\alpha \Rightarrow \emptyset$ with α as a start symbol and no productions, but that is incorrect: $\alpha \Rightarrow \emptyset$ means an empty grammar.

Example. Fig. 5 presents a simplified normalization derivation for the grammar in Fig. 3c:

$$g = \mu\,\mathrm{s}\,.(\mathrm{LPAR} \cdot (\mu\,\mathrm{ss}\,.\,\epsilon \vee \mathrm{s} \cdot \mathrm{ss}) \cdot \mathrm{RPAR}) \vee \mathrm{ATOM}$$

For space reasons, we write s for sexp, and ss for sexps. We highlight new productions generated during derivation in light gray, and omit some details via $\cdots$ for space reasons and also since normalizing tokens is straightforward; the complete derivation tree is given in the appendix in the extended version of the paper. Of particular interest is the last step, which normalizes a fixed point. In this case, s is used as the variable bound by the fixed point, and we have a non-DGNF production $\mathrm{ss} \longrightarrow \mathrm{s\,ss}$. First, s copies all productions from n_3. Then, since $\mathrm{ss} \longrightarrow \mathrm{s\,ss}$ starts with s, it expands to two productions where s is replaced by its two normal forms respectively.

3.2 Semantics of DGNF

Recall that §2.5 gave a high-level description of DGNF. This section defines the formal semantics of DGNF. We start with the expansion relation:

Definition 1 (Expansion ($G \vdash \rightsquigarrow$)). *Given a grammar G, we define the expansion relation by (1) (Base) $G \vdash n \rightsquigarrow n$; (2) (Step) if $G \vdash n \rightsquigarrow \overline{t}\,n_1\,\overline{n}$ and $(n_1 \longrightarrow N \in G)$, then $G \vdash n \rightsquigarrow \overline{t}\,N\,\overline{n}$. We write $G \vdash n \rightsquigarrow w$ when n expands to a complete word w.*

The expansion relation essentially captures what a nonterminal can expand to. For example, if $n \longrightarrow \mathrm{B}\,n_1 \in G$ and $n_1 \longrightarrow \mathrm{C} \in G$, then $G \vdash n \rightsquigarrow \mathrm{BC}$. We enforce a left-to-right expansion order for clarity and to stay close to the parsing behavior, but that is not necessary: it is easy to imagine an arbitrary order expansion, but any order leads to the same set of words.

With the notion of expansion, we define what it means for a grammar to be in DGNF precisely.

Definition 2 (Deterministic Greibach Normal Form). *A grammar G is in DGNF (i.e. it is a D grammar), if all productions are either of form $n \longrightarrow t\,\overline{n}$ or $n \longrightarrow \epsilon$, and moreover,*

- *(Determinism) for any pair of a nonterminal n and a terminal t, if there are two distinct productions $(n \longrightarrow t_1\,\overline{n}_1) \in G$ and $(n \longrightarrow t_2\,\overline{n}_2) \in G$, we have $t_1 \neq t_2$;*
- *(Guarded ϵ-productions) if $G \vdash n \rightsquigarrow \overline{t}\,n_1\,n_2\,\overline{n}$ and $(n_1 \longrightarrow \epsilon) \in G$, then for any t, either $(n_1 \longrightarrow t\,\overline{n}_1) \notin G$ or $(n_2 \longrightarrow t\,\overline{n}_2) \notin G$ for any $\overline{n}_1, \overline{n}_2$.*

The Determinism condition is straightforward, while the Guarded ϵ-productions condition needs more explanation. In §2.5, we mentioned that the ϵ-production may only be used when no terminal symbol in other productions matches the input string. Consider that the next token to match is c. The case when both the ϵ-production $n_1 \longrightarrow \epsilon$ and a production $n_1 \longrightarrow \mathrm{c}$ can match raises when n_1's follow-up nonterminal n_2 can also match c, making it possible to use the ϵ-production while $n_1 \longrightarrow \mathrm{c}$ also matches. Definition 2 captures such cases, requiring that n_1 and n_2 cannot match the same terminal if n_1 has an ϵ-production, and thus rules out example (4) in §2.5.

Now we can formally define the important property of DGNF that makes it practically useful.

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.*

3.3 Well-definedness and Correctness

Since normalization serves as the basis for the parsing algorithm, establishing its correctness is crucial for `flap`. In this section, we prove three key properties of normalization: normalization always succeeds for well-typed expressions (§3.3.1); the normalization result does not include the internal form $\alpha\,\overline{n}$ (§3.3.2); and the result of normalization is a DGNF grammar (§3.3.3).

3.3.1 Normalization is well-defined. To understand what well-definedness means, consider normalizing $g_1 \cdot g_2$. Rule *(seq)* returns $N_1\;n_2$ with $n_1 \longrightarrow N_1$ from g_1, and n_2 from g_2. However, in order for $N_1\;n_2$ to be well-formed, we must ensure that N_1 is not ϵ, or otherwise $\epsilon\;n_2$ is ill-formed. The case for sequencing is one of several places the typing information is useful. In particular, if $g_1 \cdot g_2$ is well-typed, then the typing rule for sequencing (Fig. 2) guarantees $\tau_1 \circledast \tau_2$, which says $\neg\tau_1.\textsc{Null}$. We then prove below that if an expression is not nullable, its normalization cannot have an ϵ-production. Thus N_1 cannot be ϵ, ensuring that the normalization result is in normal form.

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

With Lemma 3.2 and similar reasoning about typing for other constructs (such as alternations), we prove that normalization is well-defined for well-typed expressions.

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

3.3.2 Normalizing closed expressions produces no $\alpha\,\overline{n}$ form. Theorem 3.3 says that if an expression is well-typed then normalizing it returns a grammar G. However, G may include $n \longrightarrow \alpha\,\overline{n}$, which is not valid DGNF. In this part, we prove that normalizing *closed* well-typed expressions will not generate $\alpha\,\overline{n}$ productions. To do so, we need to reason about the occurrences of α. The following lemma says that every α returned as the head of a production must be in the typing context.

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 $\alpha \in \mathrm{dom}\,(\Gamma, \Delta)$.*

Note that the first result applies only to the start symbol n, and its proof relies on the typing rule where α is well-typed only if $\alpha \in \Gamma$ (Fig. 2). The second result applies to any n', and the most tricky case in the proof is when normalizing $\mu\alpha.\ g$, where we need to prove that the productions of the start symbol of g cannot start with α, or otherwise normalizing $\mu\alpha.\ g$ would copy all productions from g for α which would result in (e.g.) $\alpha \longrightarrow \alpha$ that fails the lemma as we are getting out of the scope of α. Fortunately, that is exactly what the first result tells us: when typing $\mu\alpha.\ g$, we add α to Δ (Fig. 2), and thus normalizing g cannot have α at the head of a production for its start symbol.

Our goal then follows as a corollary of Lemma 3.4, which says that normalizing any closed well-typed expression produces only the desired normal forms.

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* $\overline{n}$.

3.3.3 Normalization returns DGNF grammars. Finally, we prove that normalization returns DGNF grammars. That requires productions to satisfy the conditions given in Definition 2.

(1) Determinism: We prove that all $n \longrightarrow t\ \overline{n}$ for the same n start with different t. To this end, we establish the relation between starting terminals in productions and the First set of types.

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* $t \in \tau'.\textsc{First}$.

This lemma is particularly important when proving the case for normalizing $g_1 \vee g_2$, where the typing condition $\tau_1 \circledast \tau_2$ ensures that g_1 and g_2 have disjoint First, which in turn ensures that rule *(alt)* only copies distinct head terminals from g_1 and g_2.

(2) Guarded ϵ-productions: The proof is more involved, as it essentially requires us to show that during expansion $G \vdash n \rightsquigarrow \overline{t}\ n_1\ n_2\ \overline{n}$, the First set of n_1 is disjoint with the First set of n_2, if n_1 is nullable. The proof relies on showing that "expansion preserves typing". More concretely, think from the well-typed context free expressions' point of view: if $(g_1 \vee g_2) \cdot g_3$ is well-typed, then $g_1 \cdot g_3$ (and $g_2 \cdot g_3$) must also be well-typed, and going from $(g_1 \vee g_2) \cdot g_3$ to $g_1 \cdot g_3$ is one step of branching, similar to one step of expansion. We refer the reader to the appendix of the extended version of the paper for more details.

With all the conditions proved, we conclude our goal.

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.

3.4 Normalization Soundness

Our final piece of normalization metatheory establishes that normalization is sound with respect to the denotational semantics of typed context-free expressions. The denotational semantics $[\![g]\!]_\gamma$ interprets g as a language (i.e. a set of strings matched by g), where γ interprets free variables in g:

$$
\begin{array}{lcl@{\qquad}lcl}
[\![\epsilon]\!]_\gamma & = & \{\epsilon\} & [\![g_1 \cdot g_2]\!]_\gamma & = & \{w \cdot w' \mid w \in [\![g_1]\!]_\gamma \wedge w' \in [\![g_2]\!]_\gamma\} \\
[\![t]\!]_\gamma & = & \{t\} & [\![\alpha]\!]_\gamma & = & \gamma(a) \\
[\![\bot]\!]_\gamma & = & \emptyset & [\![\mu\alpha.\ g]\!]_\gamma & = & \mathsf{fix}(\lambda \mathsf{L}.[\![g]\!]_{(\gamma,\mathsf{L}/\alpha)}) \\
[\![g_1 \vee g_2]\!]_\gamma & = & [\![g_1]\!]_\gamma \cup [\![g_2]\!]_\gamma & \multicolumn{3}{l}{\mathsf{fix}(f) = \bigcup\limits_{i \in \mathbb{N}} \mathsf{L}_i \text{ where } \begin{array}{ll} \mathsf{L}_0 & = \emptyset \\ \mathsf{L}_{i+1} & = f(\mathsf{L}_i) \end{array}}
\end{array}
$$

Most cases are straightforward: ϵ denotes the singleton set containing the empty string, t the singleton containing the one-token string t, $\bot$ the empty language, and $g_1 \vee g_2$ a union of sets. The interpretation of $g_1 \cdot g_2$ appends a string from g_1 to a string from g_2. Variables α draw interpretations from the environment γ, and $\mu\alpha.\ g$ denotes the least fixed point of g with respect to α.

$$\text{fused grammar} \quad F \quad ::= \quad \{\, n \longrightarrow r\,\overline{n} \,\} \cup \{\, n \longrightarrow ?r \,\}$$

$$\begin{aligned}
\mathcal{F}[\![L, G]\!] &= \mathcal{F}_1 \cup \mathcal{F}_2 \cup \mathcal{F}_3 \\
\text{where } \mathcal{F}_1 &= \{\, n \longrightarrow r\,\overline{n} \mid r \Rightarrow \textbf{Return}\ t \in L \wedge n \longrightarrow t\,\overline{n} \in G \,\} && \textit{(inline the lexer)} \\
\mathcal{F}_2 &= \{\, n \longrightarrow r\,n \mid r \Rightarrow \textbf{Skip} \in L \wedge n \in G \,\} && \textit{(whitespace)} \\
\mathcal{F}_3 &= \{\, n \longrightarrow ?\neg r \mid n \longrightarrow \epsilon \in G \wedge r = \bigvee\{\, r \mid n \longrightarrow r\,\overline{n} \in \mathcal{F}_1 \cup \mathcal{F}_2 \,\}\} && \textit{(epsilon productions)}
\end{aligned}$$

Fig. 6. Lexer-parser fusion

To prove that our normalization is sound, we show that the normalized DGNF denotes exactly the same language as the denotation semantics of an expression. Recall that we have defined the expansion relation in Definition 1, where $G \vdash n \rightsquigarrow w$ denotes that n expands to a complete string w, where all non-terminals have been expanded. We prove the normalized grammar can expand to a string *if and only if* the string is included in the denotational semantics of the expression. The proof is done by induction first on the length of w and then on the structure of g.

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.

3.5 Implementation

The compositionality of the normalization algorithm simplifies the implementation of normalization in `flap`. 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 Fig. 4.

Unsurprisingly, 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 (αs) 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 [Reynaud et al. 2021]; if `flap` were instead implemented in a lazy language then it would be possible to implement fixed point normalization with less fuss.

4 FUSION

This section shows how `flap` fuses a separately-defined lexer and normalized parser, eliminating tokens from generated code altogether.

Canonicalizing lexer. We use *canonicalized* lexers: 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 on the right (i.e. there is exactly one Skip rule, and no token appears in more than one Return rule). Negation and intersection make it 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.

The fusion algorithm. Fig. 6 formally defines the fusion algorithm. $\mathcal{F}[\![L, G]\!]$, which operates on a canonicalized lexer L and a normalized grammar G, yielding a fused grammar F.

The fused result consists of three parts. First, we replace each production $n \longrightarrow t\,\overline{n}$ with a new production $n \longrightarrow r\,\overline{n}$, retrieving the regex r that is associated with the token t in the lexer L ($\mathcal{F}_1$). This is where the fusion function implicitly specializes the lexer to each nonterminal in the normalized grammar, and discards lexing rules that return tokens not in productions for the nonterminal. Canonicalizing the lexer to enforce disjointness simplifies this discarding of rules.

Then, we add an additional production $n \longrightarrow r\,n$ for the **skip** regex r (which may be $\bot$) for each nonterminal, allowing each nonterminal to match an arbitrary number of the skip regex ($\mathcal{F}_2$).

$$\mathcal{L}ex(L,s) = \mathcal{L}(L,\textsc{no},[\,],s)$$

$$\begin{aligned}
\mathcal{L}(L',k,rs,[\,]) &= \mathcal{M}(k,rs)\\
\mathcal{L}(L',k,rs,c{::}cs) &= \text{if } L'_c \overset{?}{=} \emptyset \text{ then } \mathcal{M}(k,rs)\\
&\quad\ \text{else case } K \text{ of } \emptyset \mapsto \mathcal{L}(L'_c,k,rs,cs)\\
&\qquad\qquad\qquad\ \ \{k'\} \mapsto \mathcal{L}(L'_c,k',cs,cs)\\
\textit{where}\ \ L'_c &= \{\partial_c(r)\Rightarrow k \mid r\Rightarrow k\in L' \wedge \partial_c(r)\neq\bot\}\\
K &= \{k \mid r\Rightarrow k\in L'_c \wedge \nu(r)\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{M}(\textsc{no},rs) &= \textsc{fail}\\
\mathcal{M}(\text{Skip},[\,]) &= [\,]\\
\mathcal{M}(\text{Skip},c{::}cs) &= \mathcal{L}(L,\textsc{no},[\,],c{::}cs)\\
\mathcal{M}(\text{Return } t,[\,]) &= [t]\\
\mathcal{M}(\text{Return } t,c{::}cs) &= t :: \mathcal{L}(L,\textsc{no},[\,],c :: cs)
\end{aligned}$$

Fig. 7. Lexing algorithm

$$\mathcal{P}arse(n\Rightarrow G,s) = \mathcal{P}(n,s)$$

$$\begin{aligned}
\mathcal{P}(n,[\,]) &= \text{if } n\longrightarrow\epsilon\in G \text{ then } [\,] \text{ else } \textsc{fail}\\
\mathcal{P}(n,t{::}ts) &= \text{if } n\longrightarrow t\overline{n}\in G \text{ then } \mathcal{Q}(\overline{n},ts)\\
&\quad\ \text{else if } n\longrightarrow\epsilon\in G \text{ then } t{::}ts \text{ else } \textsc{fail}
\end{aligned}$$

$$\begin{aligned}
\mathcal{Q}([\,],ts) &= ts\\
\mathcal{Q}(n{::}ns,ts) &= \mathcal{Q}(ns,\mathcal{P}(n,ts))
\end{aligned}$$

Fig. 8. Parsing algorithm for DGNF grammars

$$\mathcal{FP}arse(n\Rightarrow F,s) = \mathcal{G}([n],s)$$

$$\begin{aligned}
&\mathcal{F}(F_n,k,rs,s) =\\
&\quad \text{case } s \text{ of } [\,] \mapsto Step(k,rs)\\
&\qquad\qquad c{::}cs \mapsto \text{if } F'_n \overset{?}{=} \emptyset \text{ then } Step(k,rs)\\
&\qquad\qquad\qquad \text{else case } K \text{ of } \emptyset \mapsto \mathcal{F}(F'_n,k,rs,cs)\\
&\qquad\qquad\qquad\qquad\qquad\quad \{ns\} \mapsto \mathcal{F}(F'_n,\textsc{on}\ ns,cs,cs)\\
&\textit{where}\ \ F'_n = \{\langle\partial_c(r),k\rangle \mid \langle r,k\rangle\in F_n \wedge \partial_c(r)\neq\bot\}\\
&\qquad\quad\ \ K = \{k \mid \langle r,k\rangle\in F'_n \wedge \nu(r)\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{G}([\,],s) &= s\\
\mathcal{G}(n{::}ns,s) &= \mathcal{G}(ns,\mathcal{F}(F_n,k,s,s))\\
\textit{where}\ \ F_n &= \{\langle r,\overline{n}\rangle \mid n\longrightarrow r\overline{n}\in F\}\\
k &= \text{if } n \longrightarrow ?r\in F \text{ then } \textsc{back} \text{ else } \textsc{no}
\end{aligned}$$

$$\begin{aligned}
Step(\textsc{back},s) &= s\\
Step(\textsc{on}\ ns,s) &= \mathcal{G}(ns,s)\\
Step(\textsc{no},s) &= \textsc{fail}
\end{aligned}$$

Fig. 9. Parsing algorithm for fused grammars

$$\mathcal{SP}arse_{n\Rightarrow F}(s) = \mathcal{T}([n],s)$$

$$\begin{aligned}
&\mathcal{S}_{F_n,k}(rs,s) =\\
&\quad \text{case } s \text{ of } [\,] \mapsto Step(k,rs)\\
&\qquad\qquad c_i{::}cs \mapsto \text{if } F'_{n,i} \overset{?}{=} \emptyset \text{ then } Step(k,rs)\\
&\qquad\qquad\qquad \text{else case } K_i \text{ of } \emptyset \mapsto \mathcal{S}_{F'_{n,i},k}(rs,cs)\\
&\qquad\qquad\qquad\qquad\qquad\quad \{ns\} \mapsto \mathcal{S}_{F'_{n,i},\textsc{on}\ ns}(cs,cs)\\
&\qquad\qquad c_j{::}cs \mapsto \ldots\\
&\textit{where}\ \ F'_{n,i} = \{\langle\partial_{c_i}(r),k\rangle \mid \langle r,k\rangle\in F_n \wedge \partial_{c_i}(r)\neq\bot\}\\
&\qquad\qquad\ K_i = \{k \mid \langle r,k\rangle\in F'_{n,i} \wedge \nu(r)\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}([\,],s) &= s\\
\mathcal{T}(n{::}ns,s) &= \mathcal{T}(ns,\mathcal{S}_{F_n,k}(s,s))\\
\textit{where}\ \ F_n &= \{\langle r,\overline{n}\rangle \mid n\longrightarrow r\overline{n}\in F\}\\
k &= \text{if } n \longrightarrow ?r\in F \text{ then } \textsc{back} \text{ else } \textsc{no}
\end{aligned}$$

$$\begin{aligned}
Step(\textsc{back},s) &= s\\
Step(\textsc{on}\ ns,s) &= \mathcal{T}(ns,s)\\
Step(\textsc{no},s) &= \textsc{fail}
\end{aligned}$$

Fig. 10. Staged parsing algorithm

Finally, for nonterminals with an ϵ-production, the discarded regexes, along with the skip regex, are incorporated into a lookahead regex ($\mathcal{F}_3$). That is, we add a lookahead production $n \longrightarrow ?\neg r$ for the regex that is the complement of the regexes that appear in other productions for n.

Fusion with normalized grammars is strikingly simple; it would be much more involved to directly fuse context-free expressions with the lexing rules. As with normalized grammars, an expansion relation for fused grammars would guarantee that every expansion has a unique derivation.

5 IMPLEMENTATION OF PARSING

This section describes the lexing and parsing algorithms, shows how to stage the parsing algorithm to improve performance, and explains details of the implementation of the algorithms in `flap`.

5.1 The lexing algorithm

Fig. 7 presents the lexing algorithm. The algorithm has conventional *longest-match* semantics: each token returned corresponds to the rule matching the longest possible prefix of the input. This behaviour is implemented by repeatedly updating the best match seen *so far* until no rule matches.

The top-level function $\mathcal{L}ex$ takes lexing rules L and input string s. For simplicity, we assume utility functions $\mathcal{L}$ and $\mathcal{M}$ can freely access L. At a high level, $\mathcal{L}$ reads a single token from a prefix of a string, pairs the token action with the remainder of the string, and passes it to $\mathcal{M}$. $\mathcal{M}$ constructs a sequence of tokens, updating the sequence according to the action passed from $\mathcal{L}$.

$\mathcal{L}$ has four arguments: the lexing rules L'; a token action k representing the best match so far; the remainder string rs for the best match; the input string s. For empty input the best match information k and rs is passed to $\mathcal{M}$. For non-empty input $c{::}cs$, the result depends on L'_c, the lexing rules updated to use the non-empty *derivatives with respect to* c of the string. If L' is empty, lexing cannot advance, so $\mathcal{L}$ transfers control to $\mathcal{M}$. Otherwise, the result depends on the rule $r \Rightarrow a$ that matches the string up to this point including c (i.e. the rule that accepts ϵ after consuming c). If there is no such rule, then lexing continues with k. If there is such a rule, it is unique (since lexing rules are disjoint (§4)), and lexing continues with the new longest match k'.

The $\mathcal{M}$ function has two arguments: an action k, and a remainder string rs. The sentinel NO indicates that lexing has failed. For Skip, lexing continues if the remainder rs is non-empty. For Return t, t is added to the output sequence, and lexing continues if the remainder rs is non-empty. In the cases where lexing continues, it commences by supplying NO for the best-match-so-far, so that reading the next token only succeeds if $\mathcal{L}$ matches a non-empty prefix of the remaining input.

5.2 The DGNF parsing algorithm

Fig. 8 presents the parsing algorithm for DGNF grammars. Deterministic parsing makes the algorithm simple, since there is no need for backtracking.

$\mathcal{P}arse$ is the top-level parsing algorithm which takes the parsing grammar $n \Rightarrow G$ and a sequence of tokens s. There are two key functions: $\mathcal{P}$ parses using a single nonterminal n, and $\mathcal{Q}$ parses using a sequence of nonterminals ns. Again, we assume $\mathcal{P}$ and $\mathcal{Q}$ can freely access G.

$\mathcal{P}$ takes the nonterminal n and a sequence of tokens and returns the remainder of the sequence after parsing. For empty sequences parsing succeeds only if the grammar has a rule $n \longrightarrow \epsilon$. For non-empty sequences $t{::}ts$, if the grammar has a rule $n \longrightarrow t\overline{n}$, $\mathcal{P}$ consumes t and parses ts with $\mathcal{Q}$. Otherwise, parsing succeeds (consuming nothing) only if the grammar has a rule $n \longrightarrow \epsilon$.

$\mathcal{Q}$ takes a sequence of nonterminals ns and a sequence of tokens ts and parses successive prefixes of s with each nonterminal in ns.

5.3 The parsing algorithm for fused grammars

In practice, `flap` does not use separately-defined lexing and DGNF parsing algorithms, since it fuses lexing and parsing. We presented those algorithms to allow a direct comparison with the algorithm for fused grammars.

Fig. 9 shows an algorithm for parsing with fused grammars. The algorithm combines the features of the lexing algorithm (Fig. 7) and the parsing algorithm (Fig. 8): like the lexing algorithm it maintains a set of derivatives and an action and remainder string for the current *best match*; like the parsing algorithm, it keeps track of the current non-terminal.

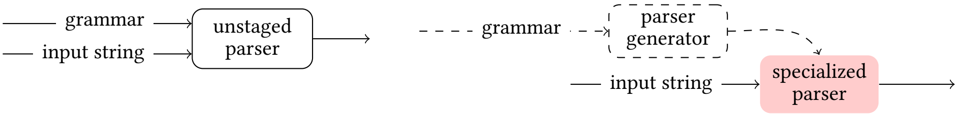
$\mathcal{F}\mathcal{P}arse$ takes the fused grammar $n \Rightarrow F$ and an input string s, with two key functions: $\mathcal{F}$ parses using a single nonterminal n, and $\mathcal{G}$ parses using a sequence of nonterminals ns using F.

$\mathcal{F}$ takes four arguments: F_n, a set of pairs representing non-epsilon productions for n; k, an action; rs, a remainder string; and s, an input string. For empty input strings the best match information k and rs is passed to $\mathcal{G}$ (via the auxiliary function $Step$). For non-empty input strings c::cs, the result depends on F'_n, the production pairs for n updated to use the non-empty *derivatives with respect to c* (§2.3) of the string. If F'_n is empty, parsing cannot proceed any further, and so $\mathcal{F}$ transfers control to $\mathcal{G}$ (via $Step$), passing the best match information. Otherwise, the result depends on the production pair $\langle r, \overline{n}\rangle$ for which r matches the string up to this point including c (i.e. the rule that accepts ϵ after consuming c). If there is no such rule, then parsing continues with k. If there is such a rule, it is unique (since the regexes for a particular nonterminal are disjoint), and it represents a new longest-match $\overline{ns}$, and parsing continues, updating the best match information to ON $\overline{ns}$. Here ON $\overline{ns}$ represents one of three continuation types, and indicates that parsing should continue using the nonterminal sequence $\overline{ns}$; the others are BACK, indicating that parsing with n should succeed, consuming no input, and NO, indicating that parsing with n should fail. The $Step$ function matches these three cases, and takes an action appropriate to each continuation.

The $\mathcal{G}$ function takes a sequence of nonterminals ns and a sequence of characters s and parses successive prefixes of s with each nonterminal in ns by calling $\mathcal{F}$. The value of $\mathcal{F}$'s k argument depends on whether there is an epsilon rule for n in the fused grammar: if so, then a parsing failure with n should backtrack, consuming no input; if not, then parsing returns FAIL.

We draw attention to two salient features of the fused parsing algorithm: first, it consists of elements from the lexing and parsing algorithms of Sections 5.1 and 5.2; second, it does not materialize the tokens produced by the lexing algorithm, instead operating directly on the character string. The final algorithm in the next section makes this even more apparent.

5.4 The staged parsing algorithm



The parsing algorithm for fused grammars described in §5.3 is practically inefficient. For each character of the input, the algorithm computes derivatives and checks emptiness and nullability for sets of regexes. However, since the regexes and other information about the grammar are known in advance of parsing, the inefficient algorithm can be *staged* [Taha 1999] to produce an efficient algorithm. The idea of staging is to identify those parts of the algorithm that do depend only on static information — i.e. on the grammar — and execute them first, leaving only the parts that depend on dynamic information — i.e. on the input string — for later. The result of staging, as illustrated in Fig. 10, is to transform the unstaged parser into a parser generator that produces as output a parser specialized to the input grammar.

Fig. 10 shows a staged version of the fused parsing algorithm. The structure of the algorithm is very close to the fused grammar parsing algorithm of §5.3: $\mathcal{S}$ corresponds to $\mathcal{F}$ and $\mathcal{T}$ corresponds to $\mathcal{G}$. However, there are three key differences.

First, those parts of the algorithm that depend on the input string are marked as *dynamic*, indicated with red highlighting. These dynamic elements are not executed immediately; instead they become part of the generated specialized parser produced by the first stage of execution.

Second, in the function $\mathcal{S}$, F_n and k have become indexes rather than arguments. Consequently, rather than being passed to the function at run-time, those arguments serve to distinguish generated functions: each instantiation of F_n and k generate a distinct function $\mathcal{S}$ in the specialized parser.

Finally, the case match in $\mathcal{S}$ is expanded to include a distinct case for each character c_i, c_j, etc. This expansion resolves a tension in the distinction between static and dynamic data: the static computation of derivatives $\partial_c(r)$ in the first stage depends on the value of c, which is only available dynamically. In the expanded case match the value of c_i is known on the right-hand side of the corresponding case, making it possible to compute derivatives valid within that program context. This scrutiny of a statically-unknown expression using a case match over its statically-known set of possible values is known as "The Trick" in the partial evaluation literature [Danvy et al. 1996].

The evaluation of the staged parsing algorithm is largely standard: the unhighlighted (static) expressions are executed first, producing the highlighted (dynamic) expressions as output. Each call to a dynamic indexed function $\mathcal{S}_{F_n,k}$ triggers the generation of a dynamic function whose body consists of the result of executing the right-hand side of $\mathcal{S}_{F_n,k}$ in Fig. 10. To ensure that the generation process terminates, the generation of these indexed functions is memoized: there is at most one generated function $\mathcal{S}_{F_n,k}$ for any particular F_n and k. The result of the algorithm is a set of mutually recursive functions that operate only on strings, not on components of the grammar:

$$\begin{array}{lcll} S_{n\to r\overline{n},\ldots,\text{BACK}}(r,s) & = & \text{case } s \text{ of} & \texttt{'a'} :: cs \mapsto S_{n\to r_a\overline{n},\text{BACK}}(r,cs) \\ & & & \texttt{'b'} :: cs \mapsto S_{n\to r_a\overline{n},\text{ON }\overline{ns}}(cs,cs) \\ & & & \ldots \\ S_{n\to r\overline{n},\ldots,\text{ON }\overline{ns}}(r,s) & = & \ldots & \end{array}$$

5.5 Implementing the staged parsing algorithm

`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).

There are three key differences between the pseudocode algorithm in Fig. 10 and `flap`'s implementation. First, while the pseudocode presents a recognizer that either consumes input or fails, `flap` supports *semantic actions* — i.e. constructing and returning ASTs or other values when parsing succeeds — as described in §2.1.

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. 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.

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"
   | _        → []
```

This excerpt shows the code generated for a single indexed function $\mathcal{S}_{F_n,k}$. There are four arguments, representing the beginning of the current token `r` (to support backtracking in the lookahead transition), the current index `i`, the input length `len`, and the input string `s`.

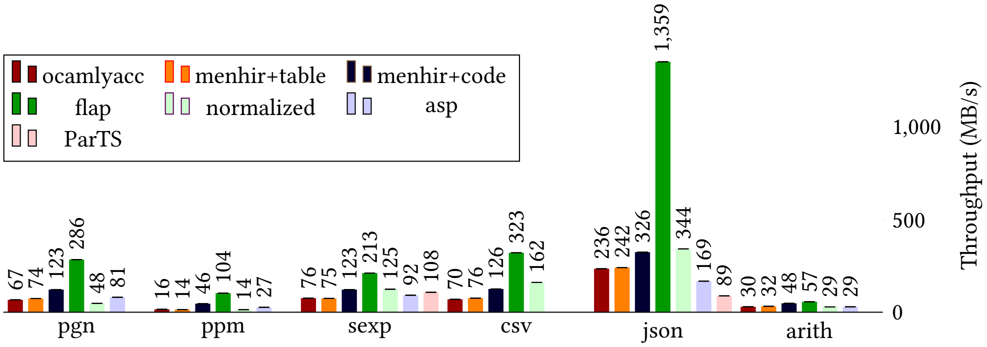


Fig. 11. Parser throughput: ocamlyacc, menhir, `flap`, asp and ParTS

The subscripts 5, 6, etc. attached to the `parse` functions correspond to the indexes F_n, k in the pseudocode algorithm; the letrec insertion library assigns a fresh subscript to each distinct index.

The character range pattern `'a'..'z'` illustrates the character class optimization described above, without which each of the characters from `'a'` to `'z'` would have a separate case in the **`match`**.

The check `i = len` determines whether `'\000'` indicates end of input or a null in the input string.

The value `[]` corresponds to a semantic action: it is the empty list returned when an empty sequence of s-expressions is parsed. It appears twice in the generated code, since (as Fig. 10 shows), parsing for a particular nonterminal can end in two ways: when it encounters the end of input, and when it encounters a non-matching character.

OCaml compiles tail calls to known functions such as $\texttt{parse}_6$ to unconditional jumps. As §6 shows, the resulting code is extremely fast.

6 EVALUATION

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

In part, `flap`'s speed is a consequence of the linear-time guarantee provided by the type system of §2.1 and by the application of staging to eliminate the overhead arising from parsing abstractions. This section shows that lexer-parser fusion provides a substantial further performance improvement by eliminating the overhead that arises from defining lexers and parsers separately, which accounts for most of the remaining running time.

Benchmarks. We compare seven implementations. All seven 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):

The parser implementations are:

(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]
(g) Parsing with normalized but unfused grammars

Implementations (a)–(c) are widely-used parser-generation tools. Implementation (d) is described in this paper. Implementations (e) and (f) are existing parser combinator libraries that guaranteee deterministic, linear-time parsing. Implementation (g) is a variant of (d) in which the grammars

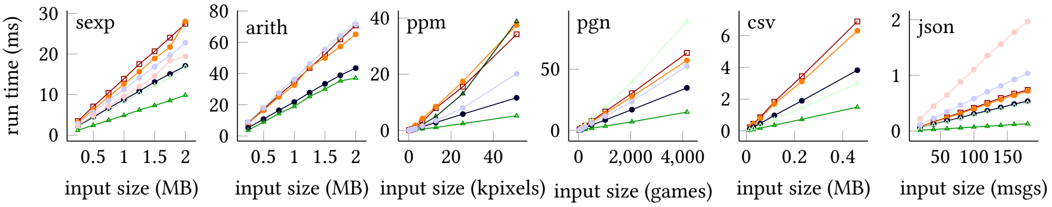


Fig. 12. Linear-time parsing (colors as Fig. 11)

used for parsing are normalized by `flap` and lexers are implemented using `flap`, but parsers and lexers are connected via OCaml's `Stream` type (as in `asp`) rather than fused together (as in `flap`).

For lexing we use `ocamllex` for (a)–(c), and the combinators supplied by each library for (d)–(g). 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)–(g) also use identically structured grammars based on the standard parser combinator interface (§2.1). However, (d)–(g) use differently-structured lexers: (e) and (f) reuse the deterministic parser combinators for lexing, while `flap` and the normalized but unfused parser use the more conventional lexing interface from Fig. 3a.

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).

The benchmarks were compiled with BER MetaOCaml N111 with flambda optimizations enabled and run on a single Intel i9-12900K core with 1GB memory running Debian Linux, using the `Core_bench` micro-benchmarking library [Hardin and James 2013].

Running time. Fig. 11 shows the throughput of the seven implementations using the benchmark grammars. Fig. 12 illustrates that all seven produce parsers with running time linear in input length.

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 1.4GB/s (a little over 2.3 cycles per byte) on the `json` benchmark. The throughput ratios of `flap` to `asp` ($\frac{286}{81} = 3.5\times$, $\frac{104}{27} = 3.9\times$, $\frac{213}{92} = 2.3\times$, $\frac{1359}{169} = 8.0\times$, $\frac{57}{29} = 2.0\times$) indicate the additional performance benefit provided by the combination of fusion and staging over staging alone. The throughput ratios of `flap` to the normalized but unfused implementation ($\frac{286}{48} = 6.0\times$, $\frac{104}{14} = 7.4\times$, $\frac{213}{125} = 1.7\times$, $\frac{1359}{344} = 4.0\times$, $\frac{57}{29} = 2.0\times$) show that the normalization step in `flap` is not sufficient to account for `flap`'s superior performance: performing lexer-parser fusion after grammar normalization provides a substantial additional speedup.

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

	Input		**Normalized**		**Fused**	**Output**
Grammar	Lex rules	CFEs	NTs	Prods	Prods	Functions
pgn	13	95	38	53	91	203
ppm	6	10	5	6	16	55
sexp	4	11	3	6	9	11
csv	3	14	5	7	7	17
json	12	42	9	33	42	93
arith	14	143	28	55	83	209

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

Compilation time (ms)
212
3.60
0.331
0.499
28.5
460

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

There are several reasons to be apprehensive about the size of code generated by `flap`. First, conversion to Greibach Normal Form is known to substantially increase grammar size; for example, in the procedure given by Blum and Koch [1999] the result of converting a grammar G has size $O(|G|^3)$. Second, fusion is inherently duplicative, repeatedly copying lexer rules into grammar productions. Finally, experience in the multi-stage programming community shows that it is easy to inadvertently generate large programs, since antiquotation makes it easy to duplicate terms.

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.

Sharing. The entries for *pgn* and *arith* hint at opportunities for further improvement. In both cases, the number of CFEs in the grammar (95 and 143) is surprisingly high, since both languages are fairly simple. Inspecting the grammar implementations 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 (§2.1) provides no 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 generated code size.

Compilation time. A final measure of practicality is the time taken to perform the fusion transformation. Slow compilation times can have a significant effect on usability; as Nielsen [1993] notes, software that takes more than ten seconds to respond can cause a user to lose focus.

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. Measurements indicate that the compilation time of the OCaml code generated by flap is also fairly low, at approximately 20ms/function, and linear in the size of the generated code.

7 RELATED WORK

Deterministic Greibach Normal Form. There are several longstanding results related to deterministic variants of Greibach Normal Form. For example, Geller et al. [1976] show that every strict deterministic language can be given a strict deterministic grammar in Greibach Normal Form, and Nijholt [1979] 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.

Combining lexers and parsers. The work most closely related to ours, by Casinghino and Roux [2020] investigates the application of traditional stream fusion techniques to parser combinators in the ParTS system. We have included their two published benchmarks in the evaluation of §6 and found that, as they report, when the flambda compiler optimizations are applied to their code, its performance is similar to the results achieved by Krishnaswami and Yallop [2019]. A major 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 fusion as a sequence of total transformations guaranteed to convert every parser into a form with good performance. More concretely, in Fig. 11, `flap` achieves two and ten times the throughputs of ParTS on the `sexp` and `json` benchmarks.

Another line of work, on *Scannerless GLR parsing* [Economopoulos et al. 2009; van den Brand et al. 2002], also aims to eliminate the boundary between lexers and parsers, but in the interface (not just in the implementation, as in `flap`). The principal aim is a principled way to handle lexical ambiguity. Scannerless parsing carries considerable cost, often running orders of magnitude slower than `flap` according to the figures given by Economopoulos et al. [2009].

Similarly, ANTLR 4 [Parr et al. 2014] supports scannerless parsing based on a top-down algorithm, ALL(*), that performs grammar analysis dynamically, during parsing. Like Scannerless GLR, it has superlinear (here $O(n^4)$) complexity in theory, but often enjoys linear performance in practice.

The *packrat* algorithm [Ford 2002] also supports a form of scannerless parsing; in contrast to Scannerless GLR and ALL(*), it is restricted to deterministic grammars. Packrat parsers are structured like backtracking recursive-descent parsers, but use lazy evaluation to construct and memoize intermediate results during parsing, reducing needless recomputation and guaranteeing linear time complexity. However, packrat has some sigificant performance limitations. Since it retains all intermediate structures, it uses space linear in the input size; further, its reported throughput (around 25 kb/second) is orders of magnitude slower than flap.

Unlike scannerless systems, `flap` does not provide a more powerful parsing interface to eliminate the need for a separate lexer. In `flap` parsers are defined using a traditional parser combinator interface and lexers are defined separately: it is only in the code generated by `flap`, not in the interface, that tokens are statically eliminated.

Context-aware scanning, introduced by Van Wyk and Schwerdfeger [2007] is another variant on the parser-lexer interface focused on disambiguation; it passes contextual information from parser to lexer about the set of valid tokens at a particular point, in a similar way to the lexer specialization in §2.7 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 parsing context.

Fusion. The notion of fusion, in the sense of merging computations to eliminate intermediate structures, has been applied in several domains, including query engines [Shaikhha et al. 2018], GPU kernels [Filipovic et al. 2015] and tree traversals [Sakka et al. 2019].

Perhaps the most widespread is stream fusion, which originated with Wadler's deforestation [Wadler 1990], and has been applied as both a traditional compiler optimization [Coutts et al. 2007] and a staged library [Kiselyov et al. 2017] with guarantees similar to `flap`'s.

Parser optimization. Finally, in contrast to the constant-time speedups resulting from lexer-parser fusion, we note an intriguing piece of work by Klyuchnikov [2010] that applies two-level-supercompilation to parser optimization, leading to asymptotic improvements.

8 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, applying the fusion techniques to more powerful parsing algorithms (e.g. LR(1)) in a traditional parser generator could make lexer-parser fusion available to many more programmers.

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 (e.g.) decompression, unicode decoding, lexing and parsing into a single computation that does not materialize intermediate values?

ACKNOWLEDGMENTS

We thank Paul Gazzillo for shepherding the paper, the anonymous reviewers for their helpful comments, and members of IFIP WG 2.11 for feedback on a presentation of this work.

This work was supported in part by a European Research Council (ERC) Consolidator Grant for the project "TypeFoundry", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 101002277), and in part by a grant from the Isaac Newton Trust (grant no. G101121).

ARTIFACT

We have made available an artifact and accompanying instructions that allow the interested reader to reproduce the claims in this paper [Yallop et al. 2023a].

REFERENCES

Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 2007. *Compilers: principles, techniques, and tools.* Vol. 2. Addison-wesley Reading.

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

Anne Brüggemann-Klein and Derick Wood. 1992. Deterministic regular languages. In *STACS 92*, Alain Finkel and Matthias Jantzen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 173–184.

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

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

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

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

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

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

Bryan Ford. 2002. Packrat parsing: : simple, powerful, lazy, linear time, functional pearl. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, Mitchell Wand and Simon L. Peyton Jones (Eds.). ACM, 36–47. https://doi.org/10.1145/581478.581483

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

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

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

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

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

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

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

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

Jakob Nielsen. 1993. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

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

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

Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) parsing: the power of dynamic analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, Andrew P. Black and Todd D. Millstein (Eds.). ACM, 579–598. https://doi.org/10.1145/2660193.2660202

François Pottier and Yann Régis-Gianas. [n. d.]. *The Menhir parser generator*. http://gallium.inria.fr/~fpottier/menhir/.

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

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

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

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

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

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

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

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.

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

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

Jeremy Yallop, Neel Krishnaswami, and Ningning Xie. 2023a. flap: A Deterministic Parser with Fused Lexing (artifact). (April 2023). https://doi.org/10.5281/zenodo.7712770

Jeremy Yallop, Ningning Xie, and Neel Krishnaswami. 2023b. flap: A Deterministic Parser with Fused Lexing. arXiv:2304.05276 [cs.PL]

Received 2022-11-10; accepted 2023-03-31