A Flexible and Efficient ML Lexer Tool based on Extended Regular Expression Submatching

Martin Sulzmann* and Pippijn van Steenhoven**

Hochschule Karlsruhe - Technik und Wirtschaft

Abstract. Lexical analysis has many applications beyond the first phase of compilation in programming language processing. We argue that extended regular expressions combined with the ability to extract submatch information significantly increase the expressiveness of lexer specifications. We show that such an expressive lexical analysis can be done efficiently using some novel automata-based methods. The approach has been implemented in an ML lexer tool which is compatible with `ocamllex`. Experimental results confirm that our approach is competitive with respect to existing ML lexer tools.

1 Introduction

The task of lexical analysis consists of identifying patterns of character sequences also known as lexeme [1]. Patterns are typically described by regular expressions. Thus, scanning can be performed efficiently by applying automata-based methods.

In this paper, we introduce an *efficient* lexical analysis approach based on *extended* regular expressions with support for intersection and negation in combined with *submatching*. As we will explain in more detail later, extended regular expressions and submatching provide the means to support clean and concise lexer specifications. While earlier works [10, 9] supports either one of the two, we are the first to support both extensions. Powerful regular expression libraries such as [11] provide also a rich feature set but can possibly exhibit a running time which can be exponential in the size of the input. Our lexical analysis approach has a guaranteed linear run-time.

Specifically, our contributions are:

- We introduce a novel and expressive scanner approach based on extended regular expressions combined with submatching (Section 2).
- The expressiveness of our approach poses new challenges when it comes to efficient scanning (Section 3). We present an efficient automata-based method to track submatches connected to extended regular expressions. Our method combines and extends prior work on partial derivative automata-based submatching [14] and partial derivatives of an extended regular expression [4] (Section 4).

* martin.sulzmann@hs-karlsruhe.de
** pip88nl@gmail.com

– We have implemented the approach in an ML lexer tool `dreml` which is compatible with `ocamllex` (Section 5).
– We present empirical measurements which show that our approach is competitive with respect to existing ML lexer tools (Section 6).

Related work is discussed in Section 3 and Section 6. Section 7 concludes. Our tool including benchmark examples is available via

`https://github.com/pippijn/dreml/`

2 Expressiveness

We start off with a cursory overview of the novel features of our lexer tool. We make use of standard math notation for regular expression patterns r:

$$r ::= \epsilon \mid \phi \mid l \in \Sigma \mid r + r \mid rr \mid r^* \mid \neg r \mid r \cap r \mid x : r$$

Letters l are taken from a finite alphabet Σ. Symbol ϵ denotes the empty word and ϕ denotes the empty language. The next forms describe alternation, concatenation and Kleene star. In examples, we write r^+ as a short-hand for rr^*. In patterns, we will write Σ as a short-hand for $l_1 + ... + l_n$ where $l_i \in \Sigma$. Choice and concatenation are assumed to be right-associative. The novelty lies in negation ($\neg$), intersection ($\cap$) and the submatch annotation $x : r$. We assume that pattern variables x are linear, i.e. occurrences are distinct.

As observed in [10], negation is useful for C comments of the form `/* ... */`. A pattern to match C comments may be written as

$$\texttt{/*}(x : \neg(\Sigma^*\texttt{*/}\Sigma^*))\texttt{*/}$$

Describing the same language without negation would require a longer, more complex and cumbersome expression:

$$\texttt{/*}((\Sigma \setminus \{*\})^*(\varepsilon + \texttt{*}^*(\Sigma \setminus \{\texttt{/}, \texttt{*}\})))^*\texttt{*/}$$

Submatch annotations are highly useful to directly extract subparts during lexical analysis to avoid clumsy post-processing steps. See the above example where we directly extract the comment text. Another typical use case for submatching are C preprocessor directives, particularly the `#include` directive. A lexical analysis is only interested in the name of the included file, which can be extracted using a pattern such as $\texttt{\#include}\ W^*\texttt{"}(x : (\Sigma \setminus \{\texttt{"}\})^*)\texttt{"}$.

Matching a valid include-directive with this pattern will record the file name in the pattern variable x. For example, consider input `#include "stdio.h"`, the resulting matching environment will consist of the set $\Gamma = \{(x : \texttt{stdio.h})\}$. The file name can then be extracted and used in a semantic action or post-processing step.

The combination of submatching and extended regular expressions is highly useful as shown by our final example. Via submatching we can specify a base pattern for C integer literals, not including hexadecimal literals:

$$r_{int} = (num : (0 \ldots 9)^+)(suf : (l + L + u + U)^*)$$

after which *num* contains the number and *suf* the type suffix. Via intersection we can restrict the pattern for octal integer literals by requiring it to begin with a zero followed by anything not containing digits 8 or 9:

$$r_{oct} = r_{int} \cap (0, (\Sigma \setminus \{8, 9\})^*)$$

In general, intersection is particularly useful in the presence of composed regular grammars. A library of standard regular expressions may define a set of valid C identifiers, which may then be restricted in specialized lexers used to verify a coding style or perform syntax highlighting based on coding conventions.

3 Efficient Submatching

Our lexer tool takes as input a sequence of patterns $(r_1, ..., r_n)$. Each r_i represents the pattern for a particular class of lexeme. The common lexical analysis approach is to seek for the longest matching pattern by testing each pattern r_i in parallel. Thus, the scanning problem can be reduced to a single pattern r. The particular challenge we face is that each r is composed of submatch annotations and extended operations such as negation and intersection. During scanning we need to efficiently keep track of submatchings.

Earlier works [6, 7] advocate the use of Thompson NFAs [15] for tracking of submatches efficiently. Roughly, the NFA non-deterministically searches for possible (sub)matchings without having to back-track. Thus, a linear running time can be guaranteed.

To deal with extended regular expressions, the Thompson transformation approach from regular expressions to NFA requires some significant changes. To deal with negation, we must first turn the underlying NFA into a DFA and then build the negation of the DFA. The DFA construction is costly and may incur some exponential explosion on the size of the automata. Similar issues arise in case of intersection where we must build the product automata. Interestingly, real world regular expression tools such as re2 [5] which rely on the Thompson NFA construction do *not* support negation and intersection (but for only very limited cases).

The work in [10] describes how to support extended regular expressions by adapting Brzozowski's derivative operation [3]. A DFA for recognizing expressions is obtained by interpreting regular expression as states. Transitions among states are obtained via the derivative operation which symbolically transforms regular expressions by taking away the leading letters. The results in [10] show that the resulting DFAs are generally optimal in size. However, like the Thompson NFA method, the Brzozowski method possibly suffers from an exponential explosion in the size of the automata. Furthermore, the work in [10] does not consider submatching which we consider a highly useful feature.

In conclusion, it is entirely possible to extend earlier works [6, 7, 10] with missing features such as submatching and extended regular expressions. However, we decide to take a different route which allows us to stick to NFAs.

To support submatching and extended regular expressions, our idea is to rely on the concept of Antimirov's partial derivatives [2]. Specifically, we build upon our own prior work [14] where we show to construct an NFA submatch automata for standard regular expressions via partial derivatives.

Like Brzozowski's derivative operation, the partial derivative operation performs a symbolic transformation on regular expressions to take away the leading letters. The difference is that Brzozowski's derivatives yield a DFA whereas Antimirov's partial derivatives yield an NFA. Roughly, the partial derivative operation takes an expression r and a letter l and yields a set of alternatives $\{r_1, ..., r_n\}$ where each r_i is a partial derivative. We find that $L(r) = L(l(r_1 + ... + r_n))$.

For example, for expression a^*a the set of partial derivatives with respect to a is $\{\epsilon, a^*a\}$. Each expression is a possible successor state. Antimirov shows that the number of partial derivatives is finite and linear in the size of the initial regular expressions. Thus, we obtain a fairly compact NFA.

Important for our work is that recently the partial derivative operation has been generalized to include additional operations such as negation and intersection [4]. As we will show in the upcoming section, we can thus extend the NFA submatch construction in [14] to the case of extended regular expressions. Experiments in the later Section 6 confirm that our approach works well in practice.

4 Extended Partial Derivative Submatch Automata

We present the details of the NFA construction for tracking submatches for an expression which may contain negation and intersection. For the construction of the automata, we use Antimirov's partial derivatives method [2] extended to the case of intersection and negation [4].

Before we dive into the technical details, we illustrate the key ideas of the construction via some example which for simplicity makes use of submatching only. For pattern $(x : a) + (y : ab)$ our construction yields the following transitions. Error states and the respective transitions are omitted for brevity.

$$\begin{aligned}
(x : a) + (y : ab) &\xrightarrow{(a,(x \to a))} (x : \epsilon) \\
(x : a) + (y : ab) &\xrightarrow{(a,(y \to a))} (y : b) \\
(y : b) &\xrightarrow{(b,(y \to b))} (y : \epsilon)
\end{aligned}$$

In the Antimirov method, NFA states can symbolically be represented by regular expressions r. There are no ϵ-transitions because the Antimirov method builds new states by taking away the leading letter. For state $(x : a) + (y : ab)$ the set of partial derivatives w.r.t. letter a is $\{(x : \epsilon), (y : b)\}$. Following [7], transitions are tagged by matchings such as $(x \to a)$ for which we use function notation.

For example, consider the transition arrow $\xrightarrow{(a,(x \to a))}$ where in case we find the input letter a we obtain the matching $(x \to a)$. Matchings are accumulated

to compute the bindings for submatch annotations. For example, running the above NFA on input word ab yields the final binding $y \to ab$.

In detail, here is a sample run of the NFA on input ab where we only follow a specific path.

$$\begin{array}{ll} & (x:a)+(y:ab) \\ \xrightarrow{(a,(y\to a))} & (y:b) \\ \xrightarrow{(b,(y\to b)\circ(y\to a))} & (y:\epsilon) \end{array}$$

Accumulation of tags is via function composition. We follow the standard definition of function composition with the exception that we concatenate the codomains of submappings with the same domain, i.e.

$$(y \to w_2) \circ (y \to w_1) = (y \to w_1 w_2)$$

Thus, we arrive at the final binding $y \to ab$.

The main challenge is to extend the partial derivative operation to the case of negation and intersection while retaining all the good properties (i.e. finite number of partial derivatives). Thankfully for us, this problem has been solved in [4]. The idea is to represent the *extended* partial derivative result as a disjunctive normal form. That is, as a set of alternatives where each alternative is a conjunction of expressions which is again represented as a set. For example, the normal form representation of $((a+b)^* \cap b^*) + c$ is $\{\{(a+b)^*, b^*\}, \{c\}\}$. In our setting, we additionally need to keep track of submatchings connected to each alternative. Hence, we need to refine the normal form in [4] to include submatching.

4.1 Extended Partial Derivatives with Submatchings

Extended regular expressions:

$$r ::= \epsilon \mid \phi \mid l \in \Sigma \mid r + r \mid rr \mid r^* \mid \neg r \mid r \cap r \mid x : r$$

Normal form representation:

$$\begin{array}{rll} \bar{r}^n & ::= \{r_1, ..., r_n\} & \text{Conjunctive clause} \\ f, g & ::= (x_1 \to w_1, ..., x_n \to w_n) & \text{Matchings} \\ \mathcal{R} & ::= (\bar{r}^n, f) & \text{Conjunctive clause with matchings} \\ \mathbb{R} & ::= \{\mathcal{R}_1, ..., \mathcal{R}_n\} & \text{Alternatives of conjunctions} \end{array}$$

Conjunctive clause to expression:

$$\begin{array}{rl} \{r\}\downarrow & = r \\ (\{r\} \cup \bar{r})\downarrow & = r \cap \bar{r}\downarrow \end{array}$$

Alternatives of conjunction to expression:

$$\begin{array}{rl} \{\} \Downarrow & = \phi \\ (\{(\bar{r}^n, f)\} \cup \mathbb{R}) \Downarrow & = \bar{r}^n\downarrow + (\mathbb{R} \Downarrow) \end{array}$$

Fig. 1. Extended Partial Derivatives Normal Form

Distributivity of concatenation, intersection and negation:

$$\mathbb{R} \odot_g r' = \{(\{rr' \,|\, r \in \bar{r}^n\}, f \circ g) \,|(\bar{r}^n, f) \in \mathbb{R}\}$$

$$\mathbb{R}_1 \mathbin{⦶} \mathbb{R}_2 = \{(\overline{r_1}^n \cup \overline{r_2}^m, f_1 \circ f_2) \,|(\overline{r_1}^n, f_1) \in \mathbb{R}_1, (\overline{r_2}^m, f_2) \in \mathbb{R}_2\}$$

$$\ominus \mathbb{R} = \begin{cases} \{(\{\neg\phi\}, id)\} & \text{if } \mathbb{R} = \{\} \\ \mathbin{⦶}_{(\bar{r}^n, f) \in \mathbb{R}} \bigcup_{r \in \bar{r}^n} (\{\neg r\}, id) & \text{otherwise} \end{cases}$$

Collection of ϵ bindings:

$$x : r\!\downarrow_\epsilon = (x \to \epsilon) \circ r\!\downarrow_\epsilon \qquad \epsilon\!\downarrow_\epsilon = id \qquad r_1 r_2\!\downarrow_\epsilon = (r_1\!\downarrow_\epsilon) \circ (r_2\!\downarrow_\epsilon)$$

$$\neg r\!\downarrow_\epsilon = id \qquad r_1 \cap r_2\!\downarrow_\epsilon = r_1\!\downarrow_\epsilon \circ r_2\!\downarrow_\epsilon$$

$$r^*\!\downarrow_\epsilon = \begin{cases} r\!\downarrow_\epsilon \text{ if } \epsilon \in L(r) \\ id \quad \text{otherwise} \end{cases} \qquad r_1 + r_2\!\downarrow_\epsilon = \begin{cases} r_1\!\downarrow_\epsilon \text{ if } \epsilon \in L(r_1) \\ r_2\!\downarrow_\epsilon \text{ if } \epsilon \in L(r_2) \end{cases}$$

Extended partial derivatives with submatching:

$$\textbf{(1)} \quad \frac{\partial}{\partial_a}(\phi) = \frac{\partial}{\partial_a}(\epsilon) = \frac{\partial}{\partial_a}(b) = \{\} \qquad \textbf{(2)} \quad \frac{\partial}{\partial_a}(a) = \{(\{\epsilon\}, id)\}$$

$$\textbf{(3)} \quad \frac{\partial}{\partial_a}(x : r) = \left\{(\{(x : \bar{r}^n\downarrow)\}, (x \to a) \circ f) \,|(\bar{r}^n, f) \in \frac{\partial}{\partial_a}(r)\right\}$$

$$\textbf{(4)} \quad \frac{\partial}{\partial_a}(r_1 + r_2) = \frac{\partial}{\partial_a}(r_1) \cup \frac{\partial}{\partial_a}(r_2)$$

$$\textbf{(5)} \quad \frac{\partial}{\partial_a}(r^*) = \frac{\partial}{\partial_a}(r) \odot_{last_{fv(r)}} r^*$$

$$\textbf{(6)} \quad \frac{\partial}{\partial_a}(r_1 r_2) = \begin{cases} \frac{\partial}{\partial_a}(r_1) \odot_{id} r_2 & \text{if } \varepsilon \notin L(r_1) \\ \frac{\partial}{\partial_a}(r_1) \odot_{id} r_2 \cup \frac{\partial}{\partial_a}(r_2) \odot_{r_1\downarrow_\epsilon} \epsilon & \text{otherwise} \end{cases}$$

$$\textbf{(7)} \quad \frac{\partial}{\partial_a}(r_1 \cap r_2) = \frac{\partial}{\partial_a}(r_1) \mathbin{⦶} \frac{\partial}{\partial_a}(r_2)$$

$$\textbf{(8)} \quad \frac{\partial}{\partial_a}(\neg r) = \ominus \frac{\partial}{\partial_a}(r)$$

Fig. 2. Extended Partial Derivatives with Submatching

Figure 1 describes the necessary adjustments. $\mathbb{R}$ describes the possible outcomes of the (shortly defined) extended partial derivative operation $\frac{\partial}{\partial_a} r$. Each component in $\mathbb{R}$ consists of a pair $(\bar{r}^n, f)$ where $\bar{r}^n$ is a set of conjunctions $\{r_1, ..., r_n\}$ and f the associated matching function (i.e. mapping of pattern variables to matched words). The translation of $\mathbb{R}$ to the underlying regular expression is straightforward. See operations $\cdot\downarrow$ and $\cdot \Downarrow$. By construction $\bar{r}^n$ is always non-empty whereas $\mathbb{R}$ can possibly be equal to the empty set. For example, consider $r = \{(\{(a+b)^*, b^*\}, f), (\{c\}, g)\}$ for which we find $r \Downarrow = ((a+b)^* \cap b^*) + c$.

Our refinement of the extended partial derivative operation $\frac{\partial}{\partial_a} r$ with submatching is given in Figure 2. We largely follow the definition given in [4] with of course some necessary adjustments due to submatching. For the definition of $\frac{\partial}{\partial_a} r$ we require auxiliary operations $\odot_g$, $\owedge$ and $\ominus$. These operations apply standard distributivity laws on expressions in normal form and additionally perform operations on matching functions.

For operation $\odot_g$, g is generally the identity function. There are two special non-identity use cases. For Kleene star, g can be customized such that we keep the matchings for all iterations or (as it is standard) only the last match. For concatenation where the first component matches ϵ, we must collect all "ϵ" bindings in combination with the operator $\downarrow_\epsilon$. Both special cases will be shortly explained in more detail. Operation $\owedge$ combines conjunctive clauses which requires us to build the composition of the associated matching functions.

Operation $\ominus$ effectively cancels any submatchings which arise below negation by simply recording the identity matching function id. The reason is that we can not give any well-defined meaning to these submatchings. For example, consider $x : \neg(y : a^*)$. Suppose the pattern matches some word. Then, pattern variable x will bind any word not containing any letter a. Clearly, the binding of y is nonsensical here because (due to the outer negation) there cannot be any match for a^*.

Next, we take a look at the various cases of the extended partial derivative operation $\frac{\partial}{\partial_a} r$. Base cases **(1)**, **(2)** are straightforward and so is case **(4)** which deals with choice.

Case **(3)** deals with submatch annotations $x : r$. The result is a set of alternatives where each conjunctive clause component $\bar{r}^n$ resulting from $\frac{\partial}{\partial_a} r$ is turned into an expression by applying $\cdot\downarrow$ to satisfy the syntactic forms of extended regular expressions. For each submatching f connected to a conjunctive clause, we compose the 'top-level' match $x \to a$ with f to build the overall submatching for each alternative in $\frac{\partial}{\partial_a}(x : r)$.

Case **(5)** deals with the Kleene star. We unfold the Kleene star once and then concatenate the result with r^*. In case of submatchings within r, the common approach is to keep only the "last" match. This is achieved via $last_{fv(r)}$ whose special purpose is to cancel all "outer" mappings connected to any variable in $fv(r)$ where $fv(r)$ refers to all pattern variables in r. [1] For example,

$$(y \to w_2) \circ last_{\{y\}} \circ (y \to w_1) = (y \to w_1)$$

For concatenation $r_1 r_2$, case **(6)**, there are two subcases depending if r_1 is nullable, i.e. $\epsilon \in L(r_1)$. The nullable test for extended regular expression is straightforward and omitted for simplicity. In case r_1 is not nullable, we only apply the partial derivative operation on r_1 and concatenate the result with r_2. The $\odot$ operation carries the identity function because the matchings for r_2 yet have to be computed.

[1] Is is also possible to tailor our approach to record the matchings for each iteration. We ignore this variation here for brevity.

If r_1 is nullable, we can simply drop r_1 and apply the partial derivative operation on r_2. What about the bindings in r_1? We clearly can not ignore them. For example, consider

$$\underbrace{((x:(y:a)^*)+(z:b^*))}_{r_1}\, r_2$$

Expression r_1 matches ϵ. This implies that the bindings of nullable subexpressions within r_1 are equal to ϵ. Both alternatives are here nullable. The left alternative $(x:(y:a)^*)$ yields $(x \to \epsilon, y \to \epsilon)$ and the right alternative yields $z \to \epsilon$. However, we will only report $(x \to \epsilon, y \to \epsilon)$ because we follow here a greedy left-most matching strategy which strictly favors *left-most* matches.

Collection of "ϵ bindings" is achieved via $r_1{\downarrow_\epsilon}$. By assumption r_1 is nullable. Hence, we recurse over the structure of r_1 and consider all submatch annotations which match ϵ. We attach the resulting bindings $r_1{\downarrow_\epsilon}$ to the bindings in $\frac{\partial}{\partial_a}(r_2)$ by slightly abusing the $\odot$ operator. The concatenated expression ϵ yields elements $r\epsilon$ in conjunctive clauses. We silently assume that $r\epsilon$ will be immediately simplified to r.

Cases **(7)** and **(8)** deal with intersection and negation and make use of the respective distributivity operators. Recall that we do not track any submatchings within negation.

4.2 Submatch NFA Construction

The construction of the actual submatch automata proceeds as follows. We repeatedly apply the $\frac{\partial}{\partial}\cdot$ operation to compute the set of all states, starting with the pattern r. This set is finite as verified in [4]. Hence, we can apply the following fixpoint construction:

$$\begin{aligned} \mathit{fix}(\{r_1,...,r_n\}) := \mathsf{let}\ & x = \{r_1,...,r_n\} \cup \textstyle\bigcup_{a\in\Sigma, r_i\in\{r_1,...,r_n\}} \\ \mathsf{in}\ & \mathsf{if}\ x = \{r_1,...,r_n\} \\ & \quad \mathsf{then}\ \{r_1,...,r_n\} \\ & \quad \mathsf{else}\ \mathit{fix}(x) \end{aligned}$$

The set $\mathit{fix}(\{r\})$ denotes the set of states of the automata resulting from r where r is the initial state and any state $r' \in \mathit{fix}(r)$ where $\epsilon \in L(r')$ is a final state.

We assume that transitions are recorded in some set T where T is defined as follows:

$$T = \{r_1 \xrightarrow{(a,f)} r_2 \mid \begin{array}{l}\text{for each } \ r_1, r_2 \in \mathit{fix}(\{r\}) \wedge a \in \Sigma \text{ where} \\ \text{for some } (\bar{r}^n, f') \in \frac{\partial}{\partial_a} r_1 \text{ we have that } f' = f \text{ and } r_2 = \bar{r}^n{\downarrow}\}\end{array}$$

We describe the execution of the submatch automata of r on some input word. Transitions operate on a configuration $\{r_{1_{f_1}},...,r_{n_{f_n}}\}$ which is a set of active states r_i attached with the so far accumulated matching function f_i. The initial configuration is $\{r_{id}\}$. For input symbol a, the derivation step from one configuration to the next is as follow:

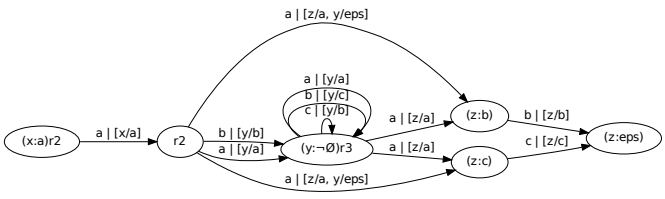
$$\{r_{1_{f_1}},...,r_{n_{f_n}}\} \xrightarrow{a} \{r'_{g\circ f_i} \mid r_{i_{f_i}} \in \{r_{1_{f_1}},...,r_{n_{f_n}}\} \wedge r_{i_{f_i}} \xrightarrow{(a,g)} r' \in T\}$$

That is, we build the set of follow states which are reachable via a transition and extend the current matching function. [2]

We may encounter duplicate states because submatching may be ambiguous. For example, consider the pattern $(x : a^*) + (y : a^*)$ where for input a we either obtain the matching $(x \to a)$ or $(y \to b)$. Following [14], we remove duplicates by giving preference to states which are to the left in the order as generated by the partial derivative operation. We assume that two expressions r_1 and r_2 are duplicates if they are syntactically equal assuming that all submatch annotations $(x : r)$ are replaced by r.

Thus, we follow the greedy left-most submatching strategy for the submatchings connected to a pattern describing a lexeme. Recall that our lexer tool guarantees to compute the longest matching among all lexeme patterns by running each pattern in parallel.

4.3 Example



Notation: `eps` $= \epsilon$ `[x/a]` $= x \to a$

Fig. 3. $r = (x : a)r_2$ where $r_2 = (y : \neg c, r_3)$ and $r_3 = (z : ab + ac)$

We consider some example to explain the construction in more detail. We assume the alphabet $\Sigma = \{a, b, c\}$ and the pattern expression $r = (x : a)r_2$ where $r_2 = (y : \neg c)r_3$ and $r_3 = (z : ab + ac)$, thus $r = (x : a)(y : \neg c)(z : ab + ac)$. The resulting NFA is given in Figure 3 where we exclude error states for brevity. Below, we consider a few steps of the extended partial derivative construction.

We start off with the initial pattern r. The computation of the extended partial derivative of r for letter a is as follows.

[2] In our informal execution notation at the beginning of this section, the extended matching is put over the derivation arrow whereas in our formalization the extended matching is now attached to the resulting state.

$$
\begin{aligned}
\tfrac{\partial}{\partial_a}(r) &= \tfrac{\partial}{\partial_a}((x:a)r_2) \\
&= \tfrac{\partial}{\partial_a}(x:a) \odot_{id} r_2 \\
&= \{(\{x:\bar{r}^n\downarrow\}, (x \to a) \circ f) \,|\, (\bar{r}^n, f) \in \tfrac{\partial}{\partial_a}(a)\} \odot_{id} r_2 \\
&\quad \text{Intermediate step:} \\
&\quad \tfrac{\partial}{\partial_a}(a) = \{(\{\varepsilon\}, id)\} \\
&= \{(\{x:\varepsilon\}, (x \to a))\} \odot_{id} r_2 \\
&= \{(\{(x:\varepsilon)r_2\}, (x \to a))\} \\
&= \{(\{r_2\}, (x \to a))\}
\end{aligned}
$$

In the last step, we apply the simplification $\epsilon r = r$. For brevity, such simplifications are omitted in the formal description in Figure 2.

Computation of $\frac{\partial}{\partial_b}(r)$ and $\frac{\partial}{\partial_c}(r)$ yield $\big\{(\{\phi r_2\}, (x \to b))\big\}$ and $\big\{(\{\phi r_2\}, (x \to c))\big\}$ which are equivalent to the error state.

We continue with the set of derived terms from the previous iteration, in this case just r_2 which is equal to $(y : \neg c)(z : ab + ac)$. We start off with building the extended partial derivative for the letter a. For the first component of the concatenated pattern $(y : \neg c)(z : ab + ac)$ we find $\epsilon \in L(\neg c)$. Hence, in the first step we apply the 'otherwise' case for concatenation. See case **(6)** in Figure 2.

$$
\begin{aligned}
&\tfrac{\partial}{\partial_a}((y:\neg c)(z:ab+ac)) \\
&= \tfrac{\partial}{\partial_a}(y:\neg c) \odot_{id} (z:ab+ac) \cup \tfrac{\partial}{\partial_a}(z:ab+ac) \odot_{y:\neg c\downarrow_\epsilon} \epsilon \\
&\quad \text{Intermediate step:} \\
&\quad \tfrac{\partial}{\partial_a}(\neg c) = \{(\{\neg\phi\}, id)\} \\
&= \{(\{(y:\neg\phi)(z:ab+ac)\}, (y \to a))\} \cup \tfrac{\partial}{\partial_a}(z:ab+ac) \odot_{y:\neg c\downarrow_\epsilon} \epsilon \\
&\quad \text{Intermediate steps:} \\
&\quad (1)\ y:\neg c\downarrow_\epsilon = (y \to \epsilon) \\
&\quad (2)\ \tfrac{\partial}{\partial_a}(z:ab+ac) = \{(\{z:b\}, (z \to a)), (\{z:c\}, (z \to a))\} \\
&\quad \text{where we simplify } \epsilon b \text{ to } b \text{ and } \epsilon c \text{ to } c \\
&\quad (3) \text{ Application of } \odot_{y:\neg c\downarrow_\epsilon} \epsilon \\
&\quad \text{invokes another simplification step, } b\epsilon \text{ to } b \text{ and } c\epsilon \text{ to } c \\
&= \{(\{(y:\neg\phi)(z:ab+ac)\}, (y \to a)), (\{z:b\}, \\
&\quad (z \to a, y \to \epsilon)), (\{z:c\}, (z \to a, y \to \epsilon))\}
\end{aligned}
$$

The remaining states and transitions are computed similarly.
Here is a sample execution for input aab.

$$
\begin{aligned}
&\{((x:a)r_2)_{id}\} \\
&\xrightarrow{a} \{(r_2)_{(x\to a)}\} \\
&\xrightarrow{a} \{((y:\neg\phi)r_3)_{(x\to a, y\to a)}, (z:b)_{(x\to a, y\to\epsilon, z\to a)}, (z:c)_{(x\to a, y\to\epsilon, z\to a)}\} \\
&\xrightarrow{b} \{((y:\neg\phi)r_3)_{(x\to a, y\to ab)}, (z:\epsilon)_{(x\to a, y\to\epsilon, z\to ab)}\}
\end{aligned}
$$

State $(z : \epsilon)$ is the only final state. Hence, the resulting matching is $(x \to a, y \to \epsilon, z \to ab)$.

5 The `dreml` Tool

Our tool aims to be a fully compatible drop-in replacement for `ocamllex` [9] with extended regular expression support and minor additional usability features. We give some examples in `dreml` syntax and discuss the current state of our implementation.

5.1 Lexer Example

We consider some of the earlier examples from Section 2 which deal with C-style comments and integer literals. Recall that both examples make use of submatching in combination with negation and intersection. Here are the examples in `dreml` syntax.

```
(* Shortcut definitions for regular expressions. *)
let digit = ['0'-'9']
let lowercase = ['a'-'z']
let suffix = ['l' 'L' 'u' 'U']
let int = (digit+ as num)(suffix+ as suf)

(* Lexer specifications. *)
rule c_token = parse
| "/*" (~(_* "*/" _*) as s) "*/"  { Comment s }
| int & ([^'0'] _*)               { IntLiteral (Decimal, num, suf) }
| int & ('0' ([^'8' '9']*)        { IntLiteral (Octal, num, suf) }
...
| _                               { failwith "invalid character" }
```

The `dreml` tool follows the `ocamllex` syntax which already has support for submatching. In addition, `dreml` adds support for negation and intersection.

- `~` for negation of regular expressions,
- `&` for their intersection,
- `re as name` to introduce a pattern variable binding `name` referring to the text matched by `re`,
- `(...)` for grouping of expressions, not introducing a pattern variable,
- `'a'` to match a single character, and
- `"abc"` as shorthand for the concatenation of characters in the string.
- `['0'-'9']` for character classes
- `[^'8' '9']` for negated character classes.

The earlier C comment text extraction is an almost literal translation to `dreml`. The earlier octal number specification

$$r_{int} = (num : (0 \ldots 9)^+)(suf : (l + L + u + U)^*)$$

is written in `dreml` syntax as follows

```
let int = (digit+ as num)(suffix+ as suf)
```

The shortcut definition `int` introduces pattern bindings `num` and `suf`. We can refer to these bindings inside the semantic actions of patterns. For example, consider

```
| int & ('0' ([^'8' '9']*)        { IntLiteral (Octal, num, suf) }
```

where on the right-hand side we refer to bindings `num` and `suf` which arise from `int`. Note that the negated character class `[^'8' '9']` corresponds to $(\Sigma \setminus \{8, 9\})^*$.

The above refines our earlier specification by including decimal numbers. Decimal numbers are required to start with a non-zero digit, since the base pattern requires at least a leading digit.

```
| int & ([^'0'] _*)               { IntLiteral (Decimal, num, suf) }
```

Readers familiar with Perl style regular expressions will notice that the `ocamllex` syntax slightly differs from Perl. The purpose of the `ocamllex` syntax is to match the OCaml syntax more closely, thus making it easier for syntax highlighting source code editors to properly display the code. Most notably the two key differences to the Perl style syntax are:

- Characters and strings must be explicitly quoted with `''` and `""`, respectively.
- The ML-style `_` operator replaces `.` as wildcard character representing Σ.

5.2 Lexer Engine

Concatenation

$$\phi r = r\phi = \phi \qquad \varepsilon r = r\varepsilon = r$$

Choice

$$r + r = r \qquad \neg\phi + r = r + \neg\phi = \neg\phi$$

Kleene star

$$r^{**} = r^* \quad \varepsilon^* = \varepsilon \quad \phi^* = \varepsilon$$

Intersection

$$r \cap r = r \qquad \phi \cap r = r \cap \phi = \phi$$

Negation

$$\neg(\neg r) = r$$

Fig. 4. Simplification rules for regular expression patterns

Simplifications To reduce the number of states during the NFA submatch automata construction, we apply simplifications on regular expression patterns.

See Figure 4. For example, via the rules for concatenation we can replace state ϕr_2 from the earlier Section 4.3 by the canonical error state ϕ. In $r + r = r$ we assume that the "right" r will be removed to maintain the greedy left-most nature of our NFA submatch engine.

Simplification rules are applied from left to right and are guaranteed to terminate as we strictly produce a smaller expression. It is straightforward to verify that simplification rules are equivalence preserving.

Character classes Currently, character classes are desugared into plain regular expressions. We plan to provide 'native' support for character classes and adopt ideas in [10] to support Unicode.

Execution The current `dreml` prototype follows an interpreter style table-driven approach. We are in the process of supporting full code generation. Our plan is to support two back-ends: a table-based one using a modified version of our prototype implementation, and a code-based back-end using mutually recursive functions. An implementation of such code generation already exists in the Thompson DFA based `re2ml` [12] tool. This older tool supports neither extended regular expressions nor pattern submatching. Our development of `dreml` will supersede this tool.

Tokenization At the time of submission, we only provide limited tokenization support because we do not fully support the `Lexing` interface in `ocamllex`. This interface abstracts processing of arbitrary streams as well as plain strings. Position information is extracted by notifying the library when matching a full lexeme. The underlying library takes care of all details concerning buffering. Hence, the implementation effort to achieve full support for tokenization is rather straightforward.

Redundancy Check Using our extended regular expression automata construction, we can decide whether the language of an expression r_1 is a subset of the language of another expression r_2. If it is, and r_1 occurs after r_2, a greedy left-most match will never reach it. We can notify the user of this problem. `re2ml` implements this check in an ad-hoc way, due to the lack of extended regular expressions. In `dreml`, we can accurately solve the equation

$$\begin{aligned} & L(r_1) \subseteq L(r_2) \\ \Leftrightarrow\ & L(r_1) \setminus L(r_2) = \emptyset \\ \Leftrightarrow\ & L(r_1) \cap \neg L(r_2) = \emptyset \\ \Leftrightarrow\ & L(r_1 \cap \neg r_2) = \emptyset \end{aligned}$$

by constructing the automata for $r_1 \cap \neg r_2$. If the resulting automata accepts no language, i.e. it is empty or contains no final state, the equation is true and we can issue a warning.

6 Empirical Results

We benchmark the performance of `dreml`. Benchmarks are executed under Ubuntu Linux 3.8.0 with 3.4GHz Intel Quad Core and 8GB RAM. Our benchmarks focus on the size of the resulting automata and the time spent on the automata construction. We also consider timing results for (sub)matching but for all cases we ignore the cost of tokenization. The contenders are `ocamllex` and `ml-ulex` which are lexing tools part of OCaml [8] and respectively SML/NJ [13]. For experiments, we use OCaml 4.00.1 and SML/NJ 110.74. `ocamllex` supports submatching and `ml-ulex` supports extended regular expressions based on the ideas described in [10]. Neither tool supports both features like our `dreml` tool.

The comparison to `ocamllex` is interesting, as we aim to produce a drop-in replacement for this tool. However, `dreml` is strictly implemented in OCaml and currently only supports a table-driven approach whereas the `ocamllex` DFA matching engine is implemented in C. Our measurements show that we already obtain good performance results.

A comparison with `ml-ulex` is more representative, since both SML/NJ and `ocamlopt`[3] produce relatively straightforward native code.

In our first benchmark, we consider a C lexical grammar specification. The `ml-ulex` and `dreml` variant make use of extended regular expressions whereas the `ocamllex` variant uses a more clumsy workaround with standard regular expressions. Both `ocamllex` and `dreml` use submatching which is not supported by `ml-ulex`.

Tool	States
`ml-ulex`	171
`ml-ulex` (minimized)	167
`ocamllex`	127
`dreml`	60

Fig. 5. Number of automata states

Figure 5 shows the number of automata states. Note that since `ocamllex` has a very low automata size limit, the grammar we use does not include keywords and simply collapses all of them into the `identifier` rule with a subsequent table lookup. As can been seen, our non-deterministic automata is the smallest (as expected). The reason why the DFA produced by `ocamllex` is smaller than the minimized `ml-ulex` DFA is unclear to us.

Figure 6 shows the timing result matching against a larger C file. Timings for `ocamllex` and `dreml` include variations where we do not perform any submatching. That is, effectively ignore the context of C comments and the path

[3] The "optimizing" OCaml native compiler merely performs some inlining, which was turned off for the tests.

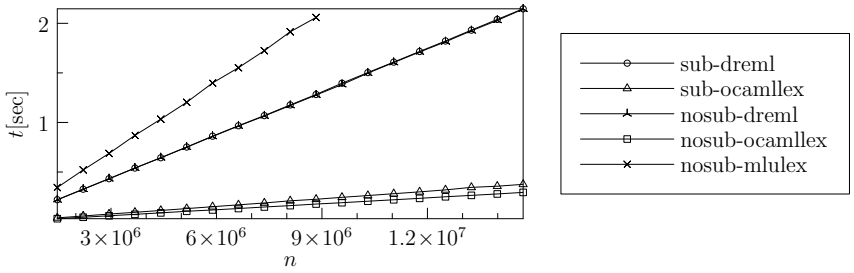


Fig. 6. Time taken lexing a Mozilla source file of n bytes

of include directives. As can been seen, for both cases performance results are comparable. This indicates that submatching generally does not incur any severe run-time penalty.

The timings for `ml-ulex` (which does not support submatching) appear to be the worst. We would have expected its timings to be similar to, or even slightly better, due to the DFA-based approach, than the ones for the NFA-based `dreml`. We suspect the 'bad' timing behavior of `ml-ulex` might be due to the fact that the input file is read in chunks of 4KB. Hence, we observe overhead due to IO.

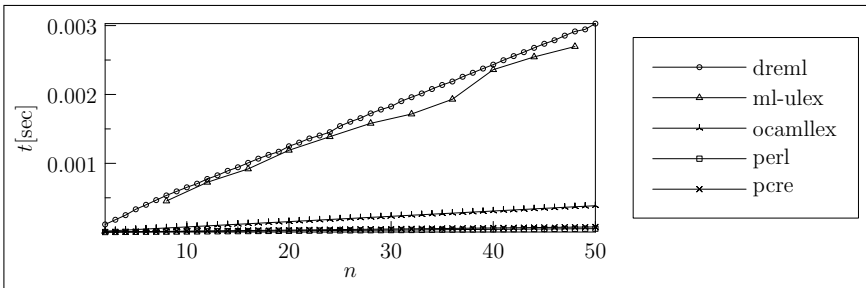


Fig. 7. Running a^* on n bytes

Figure 7 shows the timing results for matching a simple pattern against some large file. `dreml` and `ml-ulex` are comparable whereas `ocamllex` is much faster due to its C-based table engine. For comparison, we also include results for Perl and PCRE.

The next two benchmarks measure the time spent on constructing the automata. Figure 8 considers the pattern a^n which is a short-hand for a concatenated n times. Clearly, the pattern is deterministic. Hence, `dreml` will also produce a DFA. As expected, the `dreml` NFA method causes some unnecessary overhead. Interestingly, `ocamllex` performs the worst.

Figure 9 shows a worst-case scenario for DFA approaches. Performance results of `ml-ulex` and `dreml` are similar for the extended regular expression

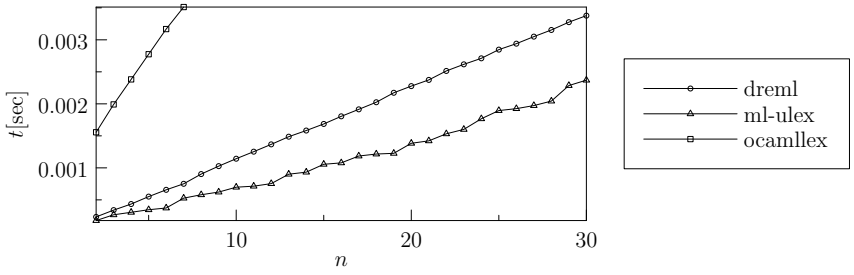


Fig. 8. Constructing automata for a^n

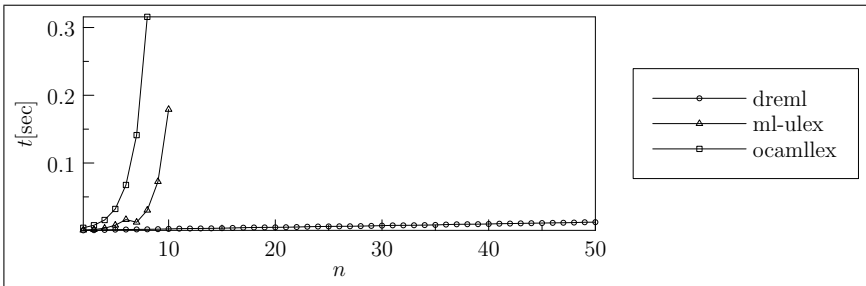


Fig. 9. Constructing automata for $(a+b)^*b(a+b)^n$

$(\neg(\neg a \cap \neg b))^*$. Obviously, we assume here that the above is not simplified to $(a+b)^*$.

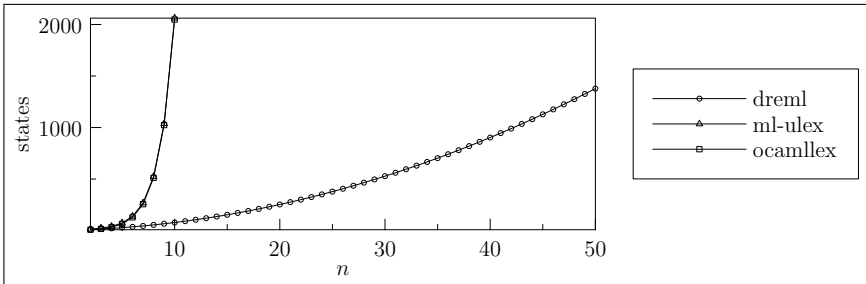


Fig. 10. Automata size for $(a+b)^*b(a+b)^n$

The exponential behavior of `ocamllex` and `ml-ulex` is due to the exponential size of the DFA automata. See Figure 10. In contrast, the NFA approach in `dreml` shows polynomial growth.

7 Conclusion

The combination of submatching and extended regular expressions improves the expressiveness of lexer specifications. Efficient lexing is achieved via a novel NFA-based method. Our prototype tool `dreml` implements the idea and can be used as a drop-in replacement for `ocamllex` with additional functionality. Initial performance results are encouraging. Future efforts will be aimed at improving usability and performance of the tool.

Some ideas for future development are:

- Add Unicode support, building on the ideas implemented in `ml-ulex` and presented in [10]. This would improve compile time performance even for non-Unicode patterns.
- Perform static analysis on regular expressions and the resulting automaton to provide better error messages, both at compile time and at runtime.
  Abstract interpretation may be helpful to prove properties of a scanner description.
- Provide an option to turn the NFA into a DFA and minimize the resulting DFA, at the expense of increased compile time. A DFA is often a feasible alternative to NFAs, when the combinatorial explosion of states does not or minimally occur.
- Implement an ML code generator producing mutually recursive functions in addition to the current table-based back-end. This is likely to vastly improve matching performance for large lexemes.
- Investigating the possibilities within a generic submatching based lexer engine.
  It would be interesting to include the semantic action functions in the AST data structure representing patterns. These functions would replace the variable names and using GADTs[4], we might be able to construct a statically typed heterogeneous matching environment. Initial attempts at this failed, so further research is required.
  This type of lexer engine would not be compatible with `ocamllex`, but would allow a user to write the semantic actions directly into the pattern in native OCaml syntax.

Acknowledgments

We thank the reviewers for their comments. We thank John Reppy and Aaron Turon for their `ml-ulex` benchmark examples.

References

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[4] Generalised Algebraic Data Types

2. Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.
3. Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.
4. Pascal Caron, Jean-Marc Champarnaud, and Ludovic Mignot. Partial derivatives of an extended regular expression. In *Proc. of LATA'11*, volume 6638 of *LNCS*, pages 179–191. Springer, 2011.
5. Russ Cox. re2 – an efficient, principled regular expression library. `http://code.google.com/p/re2/`.
6. Russ Cox. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby, ...), 2007. `http://swtch.com/~rsc/regexp/regexp1.html`.
7. Ville Laurikari. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *SPIRE*, pages 181–187, 2000.
8. OCaml. `http://caml.inria.fr/pub/docs/manual-ocaml`.
9. ocamllex. `http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual026.html`.
10. Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives reexamined. *Journal of Functional Programming*, 19(2):173–190, 2009.
11. PCRE - Perl Compatible Regular Expressions. http://www.pcre.org/.
12. re2ml: Code-based replacement for ocamllex without submatching support. `https://github.com/pippijn/re2ml`.
13. Standard ML of New Jersey. `http://www.smlnj.org/`.
14. Martin Sulzmann and Kenny Zhuo Ming Lu. Regular expression sub-matching using partial derivatives. In *Proc. of PPDP'12*, pages 79–90. ACM, 2012.
15. Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.