Regular-expression derivatives reexamined

SCOTT OWENS
University of Cambridge
Scott.Owens@cl.cam.ac.uk

JOHN REPPY
University of Chicago
jhr@cs.uchicago.edu

AARON TURON
University of Chicago
Northeastern University
turon@ccs.neu.edu

1 Introduction

The derivative of a set of strings S with respect to a symbol $\mathbf{a}$ is the set of strings generated by stripping the leading $\mathbf{a}$ from the strings in S that start with $\mathbf{a}$. For regular sets of strings, *i.e.*, sets defined by *regular expressions* (REs), the derivative is also a regular set. In a 1964 paper, Janusz Brzozowski presented an elegant method for directly constructing a recognizer from a regular expression based on *regular-expression derivatives* (Brzozowski, 1964). His approach is elegant and easily supports *extended regular expressions*; *i.e.*, REs extended with Boolean operations such as complement. Unfortunately, RE derivatives have been lost in the sands of time, and few computer scientists are aware of them.[1] Recently, we independently developed two scanner generators, one for PLT Scheme and one for Standard ML, using RE derivatives. Our experiences with this approach have been quite positive: the implementation techniques are simple, the generated scanners are usually optimal in size, and the extended RE language allows for more compact scanner specifications. Of special interest is that the implementation techniques are well-suited to functional languages that provide good support for symbolic term manipulation (*e.g.*, inductive datatypes and pattern matching).

The purpose of this paper is largely educational. Our positive experience with RE derivatives leads us to believe that they deserve the attention of the current generation of functional programmers, especially those implementing RE recognizers. We begin with a review of background material in Section 2, introducing the notation and definitions of regular expressions and their recognizers. Section 3 gives a fresh presentation of Brzozowski's work, including DFA construction with RE derivatives. In addition to reexamining Brzozowski's work, we also report on the key implementation challenges we faced in Section 4, including new techniques for handling large character sets such as Unicode (Unicode Consortium, 2003). Section 5 reports our experience in general and includes an empirical com-

[1] A quick survey of several standard compiler texts does not turn up any description of them (Aho *et al.*, 1986; Fisher & LeBlanc, Jr., 1988; Appel, 1998). The only mention we found, in fact, is an exercise in Aho and Ullman's *Theory of Parsing and Translation* (Aho & Ullman, 1972).

parison of the derivative-based scanner generator for SML/NJ with the more traditional tool it replaces. We conclude with a review of related work and a summary.

2 Preliminaries

We assume a finite alphabet Σ of symbols and use Σ^* to denote the set of all finite strings over Σ. We use a, b, c, *etc.*, to represent symbols and u, v, w to represent strings. The empty string is denoted by ε. A *language* of Σ is a (possibly infinite) set of finite strings $\mathcal{L} \subseteq \Sigma^*$.

2.1 Regular expressions

Our syntax for regular expressions includes the usual operations: concatenation, Kleene-closure, and alternation. In addition, we include the empty set ($\emptyset$) and the Boolean operations "*and*" and "*complement*."[2]

Definition 2.1
The abstract syntax of a regular expression over an alphabet Σ is given by the following grammar:

$$\begin{array}{rcll}
r, s & ::= & \emptyset & \text{empty set} \\
 & \mid & \varepsilon & \text{empty string} \\
 & \mid & a & a \in \Sigma \\
 & \mid & r \cdot s & \text{concatenation} \\
 & \mid & r^* & \text{Kleene-closure} \\
 & \mid & r + s & \text{logical or (alternation)} \\
 & \mid & r \mathbin{\&} s & \text{logical and} \\
 & \mid & \neg r & \text{complement}
\end{array}$$

These expressions are often called *extended regular expressions*, but since the extensions are conservative (*i.e.*, regular languages are closed under Boolean operations (Rabin & Scott, 1959)), we refer to them as regular expressions. Adding boolean operations to the syntax of regular expressions greatly enhances their expressiveness, as we demonstrate in Section 5.1. We use juxtaposition for concatenation and we add parentheses, as necessary, to resolve ambiguities.

The regular languages are those languages that can be described by regular expressions according to the following definition.

Definition 2.2
The *language* of a regular expression r is a set of strings $\mathcal{L}[\![r]\!] \subseteq \Sigma^*$ generated by the

[2] Other logical operations, such as exclusive or, can also be added.

following rules:

$$\begin{array}{rcl}
\mathcal{L}[\![\emptyset]\!] & = & \emptyset \\
\mathcal{L}[\![\varepsilon]\!] & = & \{\varepsilon\} \\
\mathcal{L}[\![a]\!] & = & \{a\} \\
\mathcal{L}[\![r \cdot s]\!] & = & \{u \cdot v \mid u \in \mathcal{L}[\![r]\!] \text{ and } v \in \mathcal{L}[\![s]\!]\} \\
\mathcal{L}[\![r^*]\!] & = & \{\varepsilon\} \cup \mathcal{L}[\![r \cdot r^*]\!] \\
\mathcal{L}[\![r + s]\!] & = & \mathcal{L}[\![r]\!] \cup \mathcal{L}[\![s]\!] \\
\mathcal{L}[\![r \,\&\, s]\!] & = & \mathcal{L}[\![r]\!] \cap \mathcal{L}[\![s]\!] \\
\mathcal{L}[\![\neg r]\!] & = & \Sigma^* \setminus \mathcal{L}[\![r]\!]
\end{array}$$

To avoid notational clutter, we often let an RE r denote its language $\mathcal{L}[\![r]\!]$ and refer to REs and their languages interchangeably.

2.2 Finite state machines

Finite state machines (or finite automata) provide a computational model for implementing recognizers for regular languages. For this paper, we are interested in deterministic automata, which are defined as follows:

Definition 2.3
A deterministic finite automaton (DFA) over an alphabet Σ is 4-tuple $\langle \mathcal{Q}, q_0, \mathcal{F}, \delta \rangle$, where $\mathcal{Q}$ is a finite set of *states*, $q_0 \in \mathcal{Q}$ is the distinguised *start state*, $\mathcal{F} \subseteq \mathcal{Q}$ is a set of *final* (or *accepting*) states, and $\delta : \mathcal{Q} \times \Sigma \rightarrow \mathcal{Q}$ is a partial function called the *state transition function*.

We can extend the transition function δ to strings of symbols

$$\begin{array}{rcll}
\hat{\delta}(q, \varepsilon) & = & q & \\
\hat{\delta}(q, au) & = & \hat{\delta}(q', u) & \text{when } q' = \delta(q, a) \text{ is defined}
\end{array}$$

The language accepted by a DFA is defined to be the set of strings

$$\{u \mid \hat{\delta}(q_0, u) \in \mathcal{F}\}$$

3 Regular expression derivatives

In this section, we introduce RE derivatives and show how they can be used to construct DFAs directly from REs.

3.1 Derivatives

The notion of a *derivative* applies to any language. Intuitively, the derivative of a language $\mathcal{L} \subseteq \Sigma^*$ with respect to a symbol $\mathbf{a} \in \Sigma$ is the language that includes only those suffixes of strings with a leading symbol $\mathbf{a}$ in $\mathcal{L}$.

Definition 3.1

The *derivative* of a language $\mathcal{L} \subseteq \Sigma^*$ with respect to a string $u \in \Sigma^*$ is defined to be $\partial_u \mathcal{L} = \{v \mid u \cdot v \in \mathcal{L}\}$.

For example, consider the language defined by the regular expression $r = \mathbf{ab}^*$. The derivative of r with respect to $\mathbf{a}$ is $\mathbf{b}^*$, while the derivative with respect to $\mathbf{b}$ is the empty set.

Derivatives are useful for scanner construction in part because the regular languages are closed under the derivative operation, as stated in the following theorem:

Theorem 3.1

If $\mathcal{L} \subseteq \Sigma^*$ is regular, then $\partial_u\mathcal{L}$ is regular for all strings $u \in \Sigma^*$.

Proof

We start by showing that for any $a \in \Sigma$, the language $\partial_a\mathcal{L}$ is regular. Let $\langle \mathcal{Q}, q_0, \mathcal{F}, \delta\rangle$ be a DFA that accepts the regular language $\mathcal{L}$. Then we can construct a DFA that recognizes $\partial_a\mathcal{L}$ as follows: if $\delta(q_0, a)$ is defined, then $\langle \mathcal{Q}, \delta(q_0, a), \mathcal{F}, \delta\rangle$ is a DFA that recognizes $\partial_a\mathcal{L}$ and, thus, $\partial_a\mathcal{L}$ is regular. Otherwise $\partial_a\mathcal{L} = \emptyset$, which is regular. The result for strings follows by induction. □

For regular languages that are represented as REs, there is a natural algorithm for computing the derivative as another RE. First we need a helper function, ν, from REs to REs. We say that a regular expression r is *nullable* if the language it defines contains the empty string, that is, if $\varepsilon \in \mathcal{L}[\![r]\!]$. The ν function has the property that

$$\nu(r) = \begin{cases} \varepsilon & \text{if } r \text{ is nullable} \\ \emptyset & \text{otherwise.} \end{cases}$$

and is defined as follows:

$$\begin{array}{rcl}
\nu(\varepsilon) & = & \varepsilon \\
\nu(a) & = & \emptyset \\
\nu(\emptyset) & = & \emptyset \\
\nu(r \cdot s) & = & \nu(r) \,\&\, \nu(s) \\
\nu(r + s) & = & \nu(r) + \nu(s) \\
\nu(r^*) & = & \varepsilon \\
\nu(r \,\&\, s) & = & \nu(r) \,\&\, \nu(s) \\
\nu(\neg r) & = & \left\{ \begin{array}{ll} \varepsilon & \text{if } \nu(r) = \emptyset \\ \emptyset & \text{if } \nu(r) = \varepsilon \end{array} \right.
\end{array}$$

The following rules, owed to Brzozowski, compute the derivative of a regular expression with respect to a symbol a.

$$\begin{array}{rcll}
\partial_a\,\varepsilon & = & \emptyset & \\
\partial_a\,a & = & \varepsilon & \\
\partial_a\,b & = & \emptyset & \text{for } b \neq a \\
\partial_a\,\emptyset & = & \emptyset & \\
\partial_a\,(r \cdot s) & = & \partial_a\,r \cdot s + \nu(r) \cdot \partial_a\,s & \\
\partial_a\,(r^*) & = & \partial_a\,r \cdot r^* & \\
\partial_a\,(r + s) & = & \partial_a\,r + \partial_a\,s & \\
\partial_a\,(r \mathbin{\&} s) & = & \partial_a\,r \mathbin{\&} \partial_a\,s & \\
\partial_a\,(\neg r) & = & \neg(\partial_a\,r) &
\end{array}$$

The rules are extended to strings as follows:

$$\begin{array}{rcl}
\partial_\varepsilon\,r & = & r \\
\partial_{ua}\,r & = & \partial_a\,(\partial_u\,r)
\end{array}$$

3.2 Using derivatives for RE matching

Suppose we are given an RE r and a string u and we want to determine if $u \in \mathcal{L}[\![r]\!]$. We have $u \in \mathcal{L}[\![r]\!]$ if, and only if, $\varepsilon \in \mathcal{L}[\![\partial_u\,r]\!]$, which is true exactly when $\varepsilon = \nu(\partial_u\,r)$. Combining this fact with the definition of ∂_u leads to an algorithm for testing if $u \in \mathcal{L}[\![r]\!]$. We express the algorithm in terms of the relation $r \sim u$ (r *matches* the string u), defined as the smallest relation satisfying:

$$\begin{array}{rcl}
r \sim \varepsilon & \Leftrightarrow & \nu(r) = \varepsilon \\
r \sim a \cdot w & \Leftrightarrow & \partial_a\,r \sim w
\end{array}$$

It is straightforward to show that $r \sim u$ if, and only if, $u \in \mathcal{L}[\![r]\!]$.

Notice that when an RE matches a string, we compute a derivative for each of the characters in the string. For example, consider the derivation of $\mathbf{a} \cdot \mathbf{b}^* \sim \mathbf{abb}$:

$$\begin{array}{rcl}
\mathbf{a} \cdot \mathbf{b}^* \sim \mathbf{abb} & \Leftrightarrow & \partial_{\mathbf{a}}\,\mathbf{a} \cdot \mathbf{b}^* \sim \mathbf{bb} \\
& \Leftrightarrow & \mathbf{b}^* \sim \mathbf{bb} \\
& \Leftrightarrow & \partial_{\mathbf{b}}\,\mathbf{b}^* \sim \mathbf{b} \\
& \Leftrightarrow & \mathbf{b}^* \sim \mathbf{b} \\
& \Leftrightarrow & \partial_{\mathbf{b}}\,\mathbf{b}^* \sim \varepsilon \\
& \Leftrightarrow & \mathbf{b}^* \sim \varepsilon \\
& \Leftrightarrow & \nu(\mathbf{b}^*) = \varepsilon
\end{array}$$

When the RE does not match the string, we reach a derivative that is the RE $\emptyset$, and stop.

For example,

$$\begin{aligned}
\mathbf{a}\cdot\mathbf{b}^* \sim \mathbf{aba} &\Leftrightarrow \partial_{\mathbf{a}}\,\mathbf{a}\cdot\mathbf{b}^* \sim \mathbf{ba}\\
&\Leftrightarrow \mathbf{b}^* \sim \mathbf{ba}\\
&\Leftrightarrow \partial_{\mathbf{b}}\,\mathbf{b}^* \sim \mathbf{a}\\
&\Leftrightarrow \mathbf{b}^* \sim \mathbf{a}\\
&\Leftrightarrow \partial_{\mathbf{a}}\,\mathbf{b}^* \sim \varepsilon\\
&\Leftrightarrow \emptyset \sim \varepsilon\\
&\Leftrightarrow \nu(\emptyset) = \varepsilon \qquad \text{(false)}
\end{aligned}$$

3.3 Using derivatives for DFA construction

Before describing DFA construction, we need another definition:

Definition 3.2
We say that r and s *are equivalent*, written $r \equiv s$, if $\mathcal{L}[\![r]\!] = \mathcal{L}[\![s]\!]$. We write $[r]_{\equiv}$ for the set $\{s \mid r \equiv s\}$, which is the equivalence class of r under $\equiv$.

For example, $\mathbf{a}+\mathbf{b} \equiv \mathbf{b}+\mathbf{a}$.

The matching relation gives an algorithm for testing a string against an RE by computing successive derivatives of the RE for successive characters in the string. At each step we have a residual RE that must match a residual string. If, instead of computing the derivatives on the fly, we precompute the derivative for each symbol in Σ, we can construct a DFA recognizer for the language of the RE. The states of the DFA are RE equivalence classes and the transition function is the derivative function on those classes: $\delta(q, [a]_{\equiv}) = [\partial_a\,(q)]_{\equiv}$. This function is well-defined because the derivatives of equivalent REs are equivalent. In constructing the DFA, we label each state with an RE representing its equivalence class. Accepting states are those states labeled by nullable REs, and the error state is labeled by $\emptyset$. The key challenge in making this algorithm practical is developing an efficient test for RE equivalence. We return to this point in the next section.

Figure 1 gives the complete algorithm for constructing a DFA $\langle \mathcal{Q}, q_0, \mathcal{F}, \delta\rangle$ using derivatives. The goto function constructs the transition from a state q for when the symbol c is encountered, while the explore function collects together all of the possible transitions from the state q. Together, these functions perform a depth-first traversal of the DFA's state graph while constructing it. Note that we test RE equivalence when checking to see if q_c is a new state. Brzozowski proved that an RE can have only finitely-many derivatives (up to RE equivalence), which guarantees the termination of the algorithm. Once the state graph, represented by the $(\mathcal{Q}, \delta)$ pair, has been constructed, it is simple to compute the accepting states and construct the DFA 4-tuple.

3.4 An example

Consider the RE $\mathbf{a}\cdot\mathbf{b}+\mathbf{a}\cdot\mathbf{c}$ over the alphabet $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$. The DFA construction for this RE starts with $q_0 = \partial_\varepsilon\,(\mathbf{a}\cdot\mathbf{b}+\mathbf{a}\cdot\mathbf{c}) = \mathbf{a}\cdot\mathbf{b}+\mathbf{a}\cdot\mathbf{c}$ and proceeds as follows:

1. compute $\partial_{\mathbf{a}}\,q_0 = \partial_{\mathbf{a}}\,(\mathbf{a}\cdot\mathbf{b}+\mathbf{a}\cdot\mathbf{c}) = \mathbf{b}+\mathbf{c}$, which is new, so call it q_1.

```
fun goto $q$ $(c, (\mathcal{Q}, \delta)) =$
    let $q_c = \partial_c\, q$
    in
      if $\exists q' \in \mathcal{Q}$ such that $q' \equiv q_c$
        then $(\mathcal{Q}, \delta \cup \{(q, c) \mapsto q'\})$
        else
          let $\mathcal{Q}' = \mathcal{Q} \cup \{q_c\}$
          let $\delta' = \delta \cup \{(q, c) \mapsto q_c\}$
          in explore $(\mathcal{Q}',\ \delta',\ q_c)$

and explore $(\mathcal{Q},\ \delta,\ q) =$ fold (goto $q$) $(\mathcal{Q}, \delta)\ \Sigma$

fun mkDFA $r =$
    let $q_0 = \partial_\varepsilon\, r$
    let $(\mathcal{Q}, \delta) =$ explore $(\{q_0\},\ \{\},\ q_0)$
    let $\mathcal{F} = \{q \mid q \in \mathcal{Q}$ and $\nu(q) = \varepsilon\}$
    in $\langle \mathcal{Q}, q_0, \mathcal{F}, \delta \rangle$
```

Fig. 1. DFA construction using RE derivatives

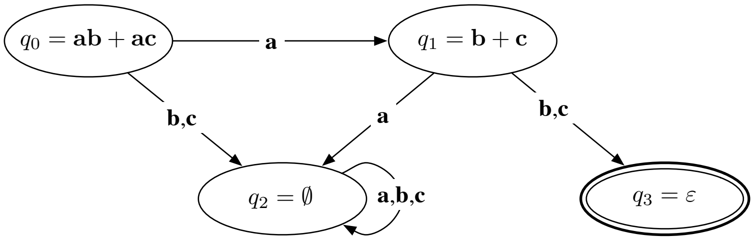

$q_0 = \mathbf{ab} + \mathbf{ac}$
a
$q_1 = \mathbf{b} + \mathbf{c}$
b,c
a
b,c
$q_2 = \emptyset$
a,b,c
$q_3 = \varepsilon$

Fig. 2. The DFA for $\mathbf{ab} + \mathbf{ac}$

2. compute $\partial_{\mathbf{a}}\, q_1 = \partial_{\mathbf{a}}\, (\mathbf{b} + \mathbf{c}) = \emptyset$, which is new, so call it q_2.
3. compute $\partial_{\mathbf{a}}\, q_2 = \partial_{\mathbf{a}}\, \emptyset = \emptyset = q_2$.
4. likewise $\partial_{\mathbf{b}}\, q_2 = q_2$ and $\partial_{\mathbf{c}}\, q_2 = q_2$.
5. compute $\partial_{\mathbf{b}}\, q_1 = \partial_{\mathbf{b}}\, (\mathbf{b} + \mathbf{c}) = (\varepsilon + \emptyset) \equiv \varepsilon$, which is new, so call it q_3.
6. compute $\partial_{\mathbf{a}}\, q_3 = \partial_{\mathbf{a}}\, \varepsilon = \emptyset = q_2$.
7. likewise $\partial_{\mathbf{b}}\, q_3 = q_2$ and $\partial_{\mathbf{c}}\, q_3 = q_2$.
8. compute $\partial_{\mathbf{c}}\, q_1 = \partial_{\mathbf{c}}\, (\mathbf{b} + \mathbf{c}) = (\emptyset + \varepsilon) \equiv \varepsilon = q_3$
9. compute $\partial_{\mathbf{b}}\, q_0 = \partial_{\mathbf{b}}\, (\mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}) = \emptyset = q_2$.
10. compute $\partial_{\mathbf{c}}\, q_0 = \partial_{\mathbf{c}}\, (\mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}) = \emptyset = q_2$.

Note that since $\nu(q_3) = \varepsilon$, q_3 is an accepting state. Figure 2 shows the resulting DFA in graphical form.

4 Practical DFA construction

While the algorithm given in Figure 1 is simple, we are faced with three issues that must be addressed to build an efficient implementation.

1. The problem of determining when two REs are equivalent, which is used to test if $q' \equiv q_c$ in the goto function, is expensive. In fact, deciding language equality for regular expressions with intersection and complement operators is of non-elementary complexity (Aho *et al.*, 1974).
2. The iteration over the symbols in Σ that is used to compute the δ function is not practical for large alphabets (*e.g.*, the Unicode character set has over 1.1 million code points).
3. A scanner generator typically takes a collection of REs as its input specification, whereas the algorithm in Figure 1 builds a DFA for a single RE.

These issues are addressed in the following three subsections.

4.1 Weaker notions of RE equivalence

The DFA construction algorithm in Figure 1 only introduces a new state when no equivalent state is present. Brzozowski proved that this check for state equivalence guarantees the minimality of the DFA produced by the algorithm, but checking RE equivalence is expensive, so in practice we change the test to

$$\exists q' \in \mathcal{Q} \text{ such that } q' \approx q_c$$

where $\approx$ is an approximation of RE equivalence that is defined as follows:

Definition 4.1
Let $\approx$ denote the least relation on REs including the following equations:

$$\begin{array}{rcl@{\qquad\qquad}crcl}
r \mathbin{\&} r & \approx & r & (*) & r + r & \approx & r \\
r \mathbin{\&} s & \approx & s \mathbin{\&} r & (*) & r + s & \approx & s + r \\
(r \mathbin{\&} s) \mathbin{\&} t & \approx & r \mathbin{\&} (s \mathbin{\&} t) & (*) & (r + s) + t & \approx & r + (s + t) \\
\emptyset \mathbin{\&} r & \approx & \emptyset & & \neg\emptyset + r & \approx & \neg\emptyset \\
\neg\emptyset \mathbin{\&} r & \approx & r & & \emptyset + r & \approx & r \\
 & & & & & & \\
(r \cdot s) \cdot t & \approx & r \cdot (s \cdot t) & & (r^*)^* & \approx & r^* \\
\emptyset \cdot r & \approx & \emptyset & & \varepsilon^* & \approx & \varepsilon \\
r \cdot \emptyset & \approx & \emptyset & & \emptyset^* & \approx & \varepsilon \\
\varepsilon \cdot r & \approx & r & & \neg(\neg r) & \approx & r \\
r \cdot \varepsilon & \approx & r & & & &
\end{array}$$

Two regular expressions r and s are *similar* if $r \approx s$ and *dissimilar* otherwise.

Theorem 4.1
If $r \approx s$ then $r \equiv s$; that is, similar REs are equivalent.

Proof
By induction on the rules defining similarity. The noninductive cases are simple algebraic consequences of Definition 2.2. □

Brzozowski proved that a notion of RE similarity including only the above rules marked with $(*)$ is enough to ensure that every RE has only a finite number of dissimilar derivatives. Hence, DFA construction is guaranteed to terminate if we use similarity as an approximation for equivalence. In our experience, including only the marked rules results in very large machines, but using the full set yields the minimal machine in most cases (see Section 5).

In our implementations, we maintain the invariant that all REs are in $\approx$-canonical form and use structural equality to identify equivalent REs. To ensure this invariant, we represent REs as an abstract type and use smart-constructor functions to build $\approx$-canonical forms. Each RE operator has an associated smart-constructor function that checks its arguments for the applicability of the $\approx$ equations. If an equation applies, the smart constructor simplifies the RE using the equation as a reduction from left to right. For example, the constructor for negation inspects its argument, and if it is of the form $(\neg r)$, the constructor simply returns r.

For the commutativity and associativity equations, we use these equivalences to sort the subterms in lexical order. We also use this lexical order to implement a functional finite map with RE keys. This map is used as the representation of the set $\mathcal{Q}$ of DFA states in Figure 1, where RE labels are mapped to states. The membership test $q_c \in \mathcal{Q}$ is just a lookup in the finite map.

4.2 Character sets

The presentation of traditional DFA construction algorithms (Aho *et al.*, 1986) involves iteration over the alphabet Σ, and the derivative-based algorithm in Figure 1 does as well. Iteration over Σ is inefficient but feasible for small alphabets, such as the ASCII character set, but for large alphabets, such as Unicode (Unicode Consortium, 2003), iteration over Σ is impractical. Since the out degree of any given state is usually much smaller than the size of the alphabet, it is advantageous to label state transitions with sets of characters. In this section, we describe an extension to Brzozowski's work that uses character sets to greatly reduce the number of derivatives that must be computed when determining the transitions from a given state.

The first step is to reformulate the abstract syntax of REs as follows:

$$\begin{array}{rcll}
r, s & ::= & \mathcal{S} & \text{where } \mathcal{S} \subseteq \Sigma \\
 & | & \varepsilon & \text{empty string} \\
 & | & r \cdot s & \text{concatenation} \\
 & | & r^* & \text{Kleene-closure} \\
 & | & r + s & \text{logical or (alternation)} \\
 & | & r \,\&\, s & \text{logical and} \\
 & | & \neg r & \text{complement}
\end{array}$$

Note that $\mathcal{S}$ covers both the empty set and single character cases from Definition 2.1, as well as character classes. The definitions of Sections 2 and 3 extend naturally to character sets.

$$\begin{array}{rcl}
\mathcal{L}[\![\mathcal{S}]\!] & = & \mathcal{S} \\
\nu(\mathcal{S}) & = & \emptyset \\
\partial_a \mathcal{S} & = & \left\{ \begin{array}{ll} \varepsilon & a \in \mathcal{S} \\ \emptyset & a \notin \mathcal{S} \end{array} \right.
\end{array}$$

As before, our implementation uses simplification to canonicalize REs involving character sets.

$$\begin{aligned} R + S &\approx T \text{ where } T = R \cup S \\ \neg S &\approx T \text{ where } T = \Sigma \setminus S \end{aligned}$$

where R, S, and T denote character sets.

As we remarked above, a given state q in a DFA will usually have many fewer distinct outgoing state transitions than there are symbols in Σ. Let $S_1, \ldots, S_n$ be a partition of Σ such that whenever $a, b \in S_i$, we have $\delta(q, a) = \delta(q, b)$ (equivalently: $\partial_a q \approx \partial_b q$). If we somehow knew the partition $S_1, \ldots, S_n$ for q in advance, we would only need to calculate one derivative per S_i when computing the transitions from q. Note that if the derivatives are distinct, then the partition is minimal. This last situation is described by the following definition:

Definition 4.2
Given an RE r over Σ and symbols $a, b \in \Sigma$, we say that $a \simeq_r b$ if and only if $\partial_a r \equiv \partial_b r$. The *derivative classes* of r are the equivalence classes $\Sigma/\simeq_r$. We write $[a]_r = \{b \mid a \simeq_r b\}$ for the derivative class of r represented by a.

For example, the derivative classes for $\mathbf{a} + \mathbf{b} \cdot \mathbf{a} + \mathbf{c}$ are $\{\mathbf{a}, \mathbf{c}\}$, $\{\mathbf{b}\}$, and $\Sigma \setminus \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$.

Whenever two symbols belong to the same derivative class for two REs, those symbols belong to the same derivative class for any combination of the REs. This insight is formalized by the following lemma:

Lemma 4.1
Let r and s be regular expressions and a and b symbols such that $a \simeq_r b$ and $a \simeq_s b$. Then the following equations hold:

- $a \simeq_{(r \cdot s)} b$
- $a \simeq_{(r + s)} b$
- $a \simeq_{(r \& s)} b$
- $a \simeq_{r^*} b$
- $a \simeq_{\neg r} b$

Proof
The proof follows from simple equational reasoning. For example,

$$\begin{aligned} \partial_a (r \cdot s) &\equiv \partial_a r \cdot s + \nu(r) \cdot \partial_a s \\ &\equiv \partial_b r \cdot s + \nu(r) \cdot \partial_b s \\ &\equiv \partial_b (r \cdot s) \end{aligned}$$

and thus $a \simeq_{(r \cdot s)} b$. The other equations follow similarly. □

We could determine the derivative classes of each state before finding any derivatives, but in general it is not possible to compute them without doing $O(|\Sigma|)$ work. Instead, we define a function $C : \text{RE} \to 2^{2^\Sigma}$ by structural recursion that computes an approximation of the derivative classes. For atomic REs, C gives an exact result:

$$\begin{aligned} C(\epsilon) &= \{\Sigma\} \\ C(\mathcal{S}) &= \{\mathcal{S}, \Sigma \setminus \mathcal{S}\} \end{aligned}$$

but compound REs are somewhat trickier. Lemma 4.1 provides guidance: if a and b are related in both $C(r)$ and $C(s)$, then they should also be related in $C(r+s)$, *etc.* Our algorithm is conservative because it assumes that *only* those symbols that are related in both $C(r)$ and $C(s)$ are related in $C(r+s)$ as specified by the following notation:

$$C(r) \wedge C(s) = \{\mathcal{S}_r \cap \mathcal{S}_s \mid \mathcal{S}_r \in C(r),\ \mathcal{S}_s \in C(s)\},$$

We can now define the remaining cases for C:

$$\begin{array}{rcl}
C(r \cdot s) & = & \begin{cases} C(r) & r \text{ is not nullable} \\ C(r) \wedge C(s) & \text{otherwise} \end{cases} \\
C(r+s) & = & C(r) \wedge C(s) \\
C(r \,\&\, s) & = & C(r) \wedge C(s) \\
C(r^*) & = & C(r) \\
C(\neg r) & = & C(r)
\end{array}$$

Consider once more the example $\mathbf{a} + \mathbf{b} \cdot \mathbf{a} + \mathbf{c}$:

$$\begin{array}{rcl}
C((\mathbf{a} + \mathbf{b} \cdot \mathbf{a}) + \mathbf{c}) & = & C(\mathbf{a} + \mathbf{b} \cdot \mathbf{a}) \wedge C(\mathbf{c}) \\
& = & (C(\mathbf{a}) \wedge C(\mathbf{b} \cdot \mathbf{a})) \wedge C(\mathbf{c}) \\
& = & (C(\mathbf{a}) \wedge C(\mathbf{b})) \wedge C(\mathbf{c}) \\
& = & (\{\{\mathbf{a}\}, \Sigma \setminus \{\mathbf{a}\}\} \wedge \{\{\mathbf{b}\}, \Sigma \setminus \{\mathbf{b}\}\}) \wedge \{\{\mathbf{c}\}, \Sigma \setminus \{\mathbf{c}\}\} \\
& = & \{\emptyset, \{\mathbf{a}\}, \{\mathbf{b}\}, \Sigma \setminus \{\mathbf{a}, \mathbf{b}\}\} \wedge \{\{\mathbf{c}\}, \Sigma \setminus \{\mathbf{c}\}\} \\
& = & \{\emptyset, \{\mathbf{a}\}, \{\mathbf{b}\}, \{\mathbf{c}\}, \Sigma \setminus \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}\}
\end{array}$$

As stated above, the exact derivative classes for this RE are $\{\mathbf{a}, \mathbf{c}\}$, $\{\mathbf{b}\}$, and $\Sigma \setminus \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$, so the approximation overpartitioned the alphabet. Nevertheless, we have reduced consideration to five symbol sets, and need only compute one derivative for each set.

The correctness of the derivative class approximation is easy to prove.

Theorem 4.2
Let r be a regular expression. Then for all $\mathcal{S} \in C(r)$ and $a \in \mathcal{S}$, we have $\mathcal{S} \subseteq [a]_r$.

Proof
By induction on the structure of r, using Lemma 4.1. □

With the approximation of derivative classes, we can modify the algorithm for DFA construction to only compute one derivative per approximate class. This version of the algorithm is shown in Figure 3.

4.3 Regular vectors

In order to use this DFA construction algorithm in a scanner generator, we need to extend it to handle multiple REs in parallel. Brzozowski recognized this problem and introduced *regular vectors* as an elegant solution.

Definition 4.3
An n-tuple of regular expressions, $\mathbf{R} = (r_1, \ldots, r_n)$, is called a *regular vector*.

```
fun goto $q$ $(\mathcal{S}, (\mathcal{Q}, \delta)) =$
    let $c \in \mathcal{S}$
    let $q_c = \partial_c\, q$
    in
      if $\exists q' \in \mathcal{Q}$ such that $q' \approx q_c$
        then $(\mathcal{Q}, \delta \cup \{(q, \mathcal{S}) \mapsto q'\})$
        else
          let $\mathcal{Q}' = \mathcal{Q} \cup \{q_c\}$
          let $\delta' = \delta \cup \{(q, \mathcal{S}) \mapsto q_c\}$
          in explore $(\mathcal{Q}', \delta', q_c)$

and explore $(\mathcal{Q}, \delta, q) =$ fold (goto $q$) $(\mathcal{Q}, \delta)$ $(C(q))$

fun mkDFA $r =$
    let $q_0 = \partial_\varepsilon\, r$
    let $(\mathcal{Q}, \delta) =$ explore $(\{q_0\}, \{\}, q_0)$
    let $\mathcal{F} = \{q \mid q \in \mathcal{Q}$ and $\nu(q) = \varepsilon\}$
    in $\langle \mathcal{Q}, q_0, \mathcal{F}, \delta \rangle$
```

Fig. 3. DFA construction using RE derivatives and character classes

Rather than labeling DFA states with REs, we now label them with a regular vectors. The transition function is still just the derivative function, where the derivative of a regular vector is defined componentwise:

$$\partial_a\,(r_1, \ldots, r_n) = (\partial_a\, r_1, \ldots, \partial_a\, r_n)$$

The definitions for accepting and error states must also be revised. A state is accepting if its regular vector contains a nullable RE. The error state is the regular vector with components all equal to the empty language, $\emptyset$. Finally, we can approximate the derivative classes of a regular vector by intersecting the approximate derivative classes of its components:

$$C(r_1, \ldots, r_n) = \bigwedge C(r_i)$$

5 Experience

We have experience with two independent implementations of RE-derivative-based scanner generators: `ml-ulex`, which is an SML scanner generator developed at the University of Chicago, and the PLT Scheme scanner generator. Both of these tools support extended REs and are being used on a regular basis.

5.1 Extended Regular Expressions

The inclusion of the complementation operator in the RE language increases its ability to express natural and concise specifications. For example, the following RE matches C-style comments, where a comment is started by the "**/***" sequence and ended by the first following "***/**" sequence (comment opening sequences are ignored inside of comments, *i.e.*, these comments do not nest):

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

The inner RE "$(\Sigma^*\texttt{*/}\Sigma^*)$" denotes the strings that contain the comment ending sequence "$\texttt{*/}$," and so its negation denotes the strings that do not contain the comment ending sequence. Thus, the entire RE denotes strings that start with the comment opening sequence and do not contain the comment ending sequence except as the last two elements. Expressing this pattern without the complement operator is more cumbersome:

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

One common use of the boolean operations on REs is to implement RE subtraction; *i.e.*, $r \mathbin{\&} \neg s$ to denote the strings in $\mathcal{L}[\![r]\!] \setminus \mathcal{L}[\![s]\!]$. For example, the DrScheme programming environment (Findler *et al.*, 2002) uses a generated lexer to interactively color the program text of PLT Scheme programs. To detect erroneous lexemes, which are highlighted in red, the the following style of regular expression is used:

$$(\mathit{idchar})^+ \mathbin{\&} \neg(\mathit{identifier} + \mathit{number})$$

where idchar is the set of characters that can appear in an identifier, $\mathit{identifier}$ is an RE matching valid identifiers, and number is an RE that matches numeric literals. The RE on the left of the "&" includes all potential bad identifiers, but it also includes valid strings, such as valid identifiers and numbers. To match just the erroneous identifiers, we subtract out the valid identifiers and numbers. In this example, the RE subtraction idiom removes the need to devise a positive definition of just the invalid lexemes. Such a definition would be exceptionally complex because of the nature of PLT Scheme's lexical syntax. For example, an identifier can start with the $\#$ character, but only when one of several specific strings immediately follow it.

5.2 DFA Size

Our experience has been that using RE derivatives is a straightforward way to generate recognizers from REs. It also turns out that the use of RE derivatives produces smaller state machines than the algorithm used by tools like `lex` and `ml-lex` (Appel *et al.*, 1994). We compared the size of the state machines generated by the `ml-lex` tool with those generated by our new `ml-ulex` tool. We also ran a DFA minimization algorithm over the state machines generated by `ml-ulex`. As test cases, we used 14 pre-existing `ml-lex` specifications for various languages, a specification for R^{5}RS Scheme (translated from PLT-Scheme), a specification for mining system logs for interesting events (translated from a Python script provided by Nick Russo), and an RE that recognizes the language L_2 (Sen & Roşu, 2003), where

$$L_k = \{u\texttt{\#}w\texttt{\#}v\texttt{\$}w \mid w \in \{\texttt{0}, \texttt{1}\}^k \text{ and } u, v \in \{\texttt{0}, \texttt{1}, \texttt{\#}\}^*\}$$

This last example requires use of the boolean operations for concise specification, so we did not test the `ml-lex` tool on it. The results are presented in Table 1.[3] In most cases, the

[3] We adjusted the number of states reported by `ml-lex` downward by 2, because it includes the error state and a redundant initial state in its count, whereas `ml-ulex` reports only the non-error states.

Table 1. *Number of states (best results in **bold**)*

Lexer	`ml-lex`	`ml-ulex`	Minimal	Description
Burg	61	**58**	**58**	A tree-pattern match generator
CKit	122	**115**	**115**	ANSI C lexer
Calc	**12**	**12**	**12**	Simple calculator
CM	153	**146**	**146**	The SML/NJ compilation manager
Expression	**19**	**19**	**19**	A simple expression language
FIG	150	**144**	**144**	A foreign-interface generator
FOL	**41**	**41**	**41**	First-order logic
HTML	52	**49**	**49**	HTML 3.2
MDL	161	**158**	**158**	A machine-description language
ml-lex	121	**116**	**116**	The `ml-lex` lexer
Scheme	324	**194**	**194**	R^{5}RS Scheme
SML	251	**244**	**244**	Standard ML lexer
SML/NJ	169	**158**	**158**	SML/NJ lexer
Pascal	60	**55**	**55**	Pascal lexer
ml-yacc	100	**94**	**94**	The `ml-yacc` lexer
Russo	4803	3017	**2892**	System-log data mining
L_2	*n/a*	147	**106**	Monitoring stress-test

RE derivatives method produced a smaller state machine. Most of the time, the difference is small, but in two cases (Scheme and Russo), the `ml-ulex` DFAs have a third fewer states. Furthermore, `ml-ulex` produces the minimal state machine for every example except Russo, where the DFA is 4% larger than optimal, and L_2, where the DFA is 39% larger. In both of these cases, `ml-lex` did significantly worse.

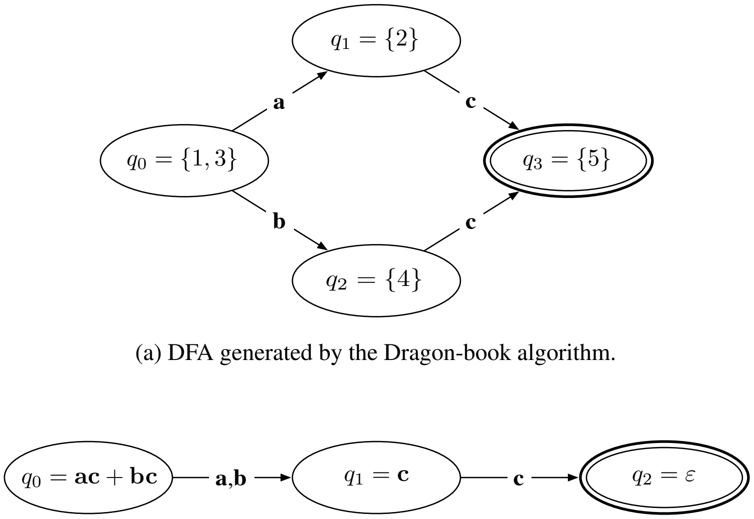
The reason that the derivative approach produces smaller machines can be illustrated using a small example, but first we must give a quick description of the algorithm used by ml-lex. This algorithm was invented by McNaughton and Yamada (McNaughton & Yamada, 1960) and is described in the "Dragon Book" (Aho *et al.*, 1986). It directly translates the abstract syntax tree (AST) representation of an RE to a DFA. The non-ε leaves in the AST are are annotated with unique positions and sets of positions are used to represent the DFA states. Intuitively, if a_i is a symbol in the RE and i is in a state q, then there is a non-error transition from q on a in the DFA. The state transition from a state q on the symbol a s computed by

$$\bigcup \text{Follow}(i) \text{ such that } i \in q \text{ and } a_i \text{ is in the RE}$$

where $\text{Follow}(i)$ is the set of positions that can follow a_i in a string matched by the RE. We demonstrate this algorithm on the following RE, which also illustrates why the derivative algorithm produces smaller DFAs:

$$(\mathbf{a}_1\mathbf{c}_2 + \mathbf{b}_3\mathbf{c}_4)\$_5$$

Here we have annotated each symbol with its position and denoted the position at the end of the RE by $\$_5$. The initial state is $q_0 = \{1, 3\}$. The construction of the DFA proceeds as follows:



(a) DFA generated by the Dragon-book algorithm.

(b) DFA generated by the derivative algorithm.

Fig. 4. DFAs for $(\mathbf{ac} + \mathbf{bc})$.

1. compute $\delta(q_0, \mathbf{a}) = \{2\}$, which is new, so call it q_1.
2. compute $\delta(q_0, \mathbf{b}) = \{4\}$, which is new, so call it q_2.
3. compute $\delta(q_1, \mathbf{c}) = \{5\}$, which is new, so call it q_3.
4. compute $\delta(q_2, \mathbf{c}) = \{5\}$, which is q_3.

This construction produces the four-state DFA shown in Figure 4(a).[4]

Now consider building a DFA for this RE using the derivative algorithm. The first state is $q_0 = \partial_\varepsilon\, \mathbf{ac} + \mathbf{bc} = \mathbf{ac} + \mathbf{bc}$.

1. compute $\delta(q_0, \mathbf{a}) = \partial_{\mathbf{a}}\, (\mathbf{ac} + \mathbf{bc}) = \mathbf{c}$, which is new, so call it q_1.
2. compute $\delta(q_0, \mathbf{b}) = \partial_{\mathbf{b}}\, (\mathbf{ac} + \mathbf{bc}) = \mathbf{c} = q_1$.
3. compute $\delta(q_1, \mathbf{c}) = \partial_{\mathbf{c}}\, \mathbf{c} = \varepsilon$, which is new, so call it q_2.

This construction produces the smaller, three-state, DFA shown in Figure 4(b). As can be seen from this example, the use of positions in the Dragon-book algorithm causes equivalent states (*i.e.*, q_1 and q_2 in the example) to be distinguished, whereas the use of canonical REs to label the states in the derivative algorithm allows their equivalence to be detected.

5.3 Effectiveness of character classes

We also used the above suite of lexer specifications to measure the usefulness of character classes. For a DFA with n states and m distinct state transitions, one has to compute at least m but no more than $n|\Sigma|$ derivatives. We instrumented `ml-ulex` to count the number of distinct state transitions and the number of approximate character classes computed by our algorithm. In all but two cases (Scheme and L_2), the approximation was perfect. In

[4] For this exercise, we are ignoring the error state.

the two cases where it was not perfect, our algorithm computed 5.4% and 6.2% more derivatives than necessary. What is more impressive is the number of derivatives that we avoid computing. If we assume the 7-bit ASCII character set as our input alphabet, then our algorithm computes only 2–4% of the possible derivatives. Thus, we conclude that character classes provide a significant benefit in the construction of DFAs, even when the underlying alphabet is small.

6 Related Work

Regular expression derivatives have been occasionally used to perform on-the-fly RE matching (without building automata) in XML validation tasks (English, 1999; Schmidt, 2002). Other than our systems, we know of at least two uses of derivatives in DFA construction. The first two versions of the Esterel language used derivatives, but the approach was abandoned in 1987 as too memory intensive (Berry, 1999); furthermore, the REs and DFAs were not used for lexical analysis. More recently, Sen and Roşu used RE derivatives to construct DFAs for program trace monitoring (Sen & Roşu, 2003). Their system generates minimal DFAs by testing full RE equivalence, using a technique called *circular coinduction*. This approach seems less practical than the approximate equivalence testing of our systems: for example, they report that computing the optimal DFA for the L_2 RE mentioned in the previous section took 18 minutes, whereas `ml-ulex` takes less than a second to compute a DFA that has only 40% more states than the optimal machine. The slowness of their approach may be owed to the fact that their method is based on rewriting, since even if we apply state minimization to this example, `ml-ulex` still takes less than a second to construct the optimal DFA.

Derivatives have largely been ignored by the scanning literature. One exception is a paper by Berry and Sethi (Berry & Sethi, 1986) that shows how a derivative-based algorithm for DFA construction can be used to derive the McNaughton and Yamada (*a.k.a.* Dragon-book) algorithm (McNaughton & Yamada, 1960). The key difference between their work and Brzozowski's derivatives algorithm is that they mark each symbol in the RE with a unique subscript. These subscripts mean that states that Brzozowski's algorithm would conflate are instead distinguished as illustrated in Figure 4. Ken Thompson, in his seminal paper on regular-expression matching (Thompson, 1968), claims

> In the terms of Brzozowski, this algorithm continually takes the left derivative of the given regular expression with respect to the text to be searched.

This claim is true if one is computing derivatives for REs where occurrences of symbols have been marked to distinguish them, but not if one is using Brzozowski's algorithm. Again, the example from Figure 4 can be used to illustrate this difference.

Berry and Sethi observed that the unmarking homomorphism does not commute with RE complement and intersection (Berry & Sethi, 1986), so algorithms based on marked symbols (*e.g.*, the Dragon-book algorithm) cannot be easily modified to support these operations. On the other hand, since the complement of a DFA is simple to compute, the standard NFA to DFA construction can be extended to support RE complements. When the algorithm encounters a complemented RE $\neg r$, it builds a NFA for r as usual, then converts the NFA to a DFA, which can be simply complemented and converted back to an NFA.

The algorithm then proceeds as usual. The lexer generator for the DMS system (Baxter *et al.*, 2004), supports complement in exactly this way.[5] We are unaware of any other lexer generators that support the complement operator.

7 Concluding remarks

In this paper, we have presented RE derivatives, which are an old, but largely forgotten, technique for constructing DFAs directly from REs. Our experience has been that RE derivatives are a superior technique for generating scanners from REs and they should be in the toolkit of any programmer. Specifically, RE derivatives have the following advantages:

- They provide a direct RE to DFA translation that is well suited to implementation in functional languages.
- They support extended REs almost for free.
- The generated scanners are often optimal in the number of states and are uniformly better than those produced by previous tools.

In addition to presenting the basic RE to DFA algorithm, we have also discussed a number of practical issues related to implementing a scanner generator that is based on RE derivatives, including supporting large character sets.

Acknowledgments

Comments from Russ Cox and the anonymous referees were quite helpful in improving the presentation. Aaron Turon's work on `ml-ulex` at the University of Chicago was supported, in part, by NSF Grant *CRI: Standard ML Software Infrastructure*.

References

Aho, Alfred V., & Ullman, Jeffrey D. (1972). *The Theory of Parsing, Translation, and Compiling*. Vol. 1. Englewood Cliffs, NJ: Prentice-Hall.

Aho, Alfred V., Hopcroft, John E., & Ullman, Jeffrey D. (1974). *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison Wesley.

Aho, Alfred V., Sethi, Ravi, & Ullman, Jeffry D. (1986). *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison Wesley.

Appel, Andrew W. (1998). *Modern Compiler Implementation in ML*. Cambridge, UK: Cambridge University Press.

Appel, Andrew W., Mattson, James S., & Tarditi, David R. 1994 (Oct.). *A lexical analyzer generator for Standard ML*. Available from `http://smlnj.org/doc/ML-Lex/manual.html`.

Baxter, Ira, Pidgeon, Christopher, & Mehlich, Michael. (2004). DMS: Program transformations for practical scalable software evolution. *International Conference on Software Engineering*.

Berry, Gérard. (1999). *The Esterel v5 Language Primer Version 5.21 release 2.0*. `ftp://ftp-sop.inria.fr/meije/esterel/papers/primer.pdf`.

Berry, Gérard, & Sethi, Ravi. (1986). From regular expressions to deterministic automata. *Theoretical Compter Science*, Dec., 117–126.

[5] Personal correspondence, Michael Mehlich, May 5, 2004

Brzozowski, Janusz A. (1964). Derivatives of regular expressions. *Journal of the ACM*, **11**(4), 481–494.

English, Joe. (1999). *How to validate XML*. `http://www.flightlab.com/~joe/sgml/validate.html`.

Findler, Robert Bruce, Clements, John, Flanagan, Cormac, Flatt, Matthew, Krishnamurthi, Shriram, Steckler, Paul, & Felleisen, Matthias. (2002). DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, **12**(2), 159–182.

Fisher, Charles N., & LeBlanc, Jr., Richard J. (1988). *Crafting a Compiler*. Menlo Park, CA: Benjamin/Cummings.

McNaughton, R., & Yamada, H. (1960). Regular expressions and state graphs for automata. *IEEE Transactions on Electronic Computers*, **9**, 39–47.

Rabin, M. O., & Scott, D. (1959). Finite automata and their decision problems. *IBM Journal of Research and Development*, **3**(2), 114–125.

Schmidt, Martin. (2002). *Design and Implementation of a Validating XML Parser in Haskell*. Masters thesis, University of Applied Sciences Wedel, Computer Science Department.

Sen, Koushik, & Roşu, Grigore. (2003). Generating optimal monitors for extended regular expressions. *Proceedings of runtime verification (RV'03)*. Electronic Notes in Theoretical Computer Science, vol. 89, no. 2. Elsevier Science.

Thompson, Ken. (1968). Regular expression search algorithm. *Communications of the ACM*, **11**(6), 419–422.

Unicode Consortium. (2003). *The Unicode Standard, Version 4*. Reading, MA: Addison-Wesley Professional.