

ML-Flex Implementation Notes

Aaron Turon
adrassi@uchicago.edu

February 17, 2006

Contents

1	Organization	2
2	Theory	3
2.1	Regular expressions	3
2.2	Derivatives	3
2.3	Factorings	5
3	Code	8
3.1	RegExp	8
3.2	LexSpec and LexOutputSpec	10
3.3	LexGen	12
3.4	Front ends	12
3.5	Back ends	12
3.6	Main	13

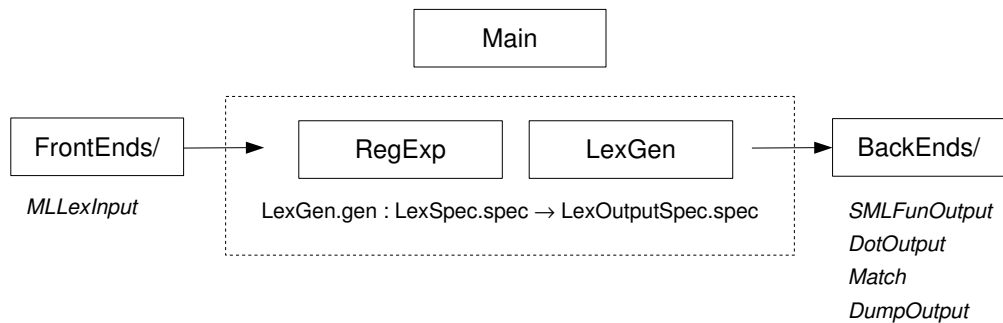


Figure 1: ML-Flex organization

1 Organization

ML-Flex is a scanner generator written in Standard ML. It replaces the older ML-Lex tool. For information about features and usage, see the ML-Flex Release Notes. This document describes the algorithms and code that make up ML-Flex.

ML-Flex is organized much like a compiler: there is a replaceable front end for parsing the ML-Lex specification format, several back ends to support various output formats, and a middle component responsible for DFA generation. The `Main` module drives the tool, while the `RegExp` and `LexGen` modules together provide DFA generation (see figure 1). The DFA generation algorithm used in ML-Flex is somewhat nonstandard; it is based on Brzozowski's notion of regular expression derivatives [Brz64]. Section 2 describes the algorithm, as well as modifications necessary to support unicode. Section 3 gives more concrete details about the code, broken down by module.

2 Theory

2.1 Regular expressions

Throughout this section, we assume an *alphabet* Σ ; any $a \in \Sigma$ is a *symbol*. Since we support unicode, Σ can be quite large. Our abstract regular expression (RE) language is as follows:

RE ::=	ϵ	empty string
	\mathcal{S}	symbol set, $\mathcal{S} \subseteq \Sigma$
	RE · RE	concatenation
	RE*	Kleene-closure
	RE RE	alternation (union)
	RE & RE	intersection
	¬RE	negation

Note that we treat symbol sets (*i.e.*, character classes) as primitive; this matches the implementation strategy and simplifies the description of DFA generation. With this representation, the empty set \emptyset and the alphabet Σ are both treated as symbol sets. The former will yield an RE that matches no input (*i.e.*, $\mathcal{L}[\emptyset] = \emptyset$), and the latter will match any single symbol. Notice also that our language of REs allows for intersection and negation in addition to the standard operations.

The semantics of our RE language are given in the form of a function $\mathcal{L}[-] : \text{RE} \rightarrow \Sigma^*$ from REs to their corresponding language over Σ :

$$\begin{aligned}\mathcal{L}[\epsilon] &= \epsilon \\ \mathcal{L}[\mathcal{S}] &= \mathcal{S} \\ \mathcal{L}[r \cdot s] &= \mathcal{L}[r] \cdot \mathcal{L}[s] \\ \mathcal{L}[r^*] &= \epsilon \cup \mathcal{L}[r] \cdot \mathcal{L}[r^*] \\ \mathcal{L}[r \mid s] &= \mathcal{L}[r] \cup \mathcal{L}[s] \\ \mathcal{L}[r \& s] &= \mathcal{L}[r] \cap \mathcal{L}[s] \\ \mathcal{L}[\neg r] &= \Sigma \setminus \mathcal{L}[r]\end{aligned}$$

2.2 Derivatives

Brzowski introduced *derivatives* of regular expressions as an alternative means of DFA construction [Brz64]. His approach is attractive because it easily allows the language of REs to be extended with arbitrary boolean operations. Further, it is intuitive, relatively easy to implement, goes directly from an RE to a DFA, and with some care in implementation can be made competitive with other DFA construction approaches. We begin by introducing the notion of a derivative of some language \mathcal{L} .

Definition. The **derivative** of a set of symbol sequences $\mathcal{L} \subset \Sigma^*$ with respect to a finite symbol sequence u is defined to be $D_u(\mathcal{L}) = \{v \mid u \cdot v \in \mathcal{L}\}$.

Derivatives give a very natural algorithm for DFA construction. Before giving that algorithm, however, we need a means of computing derivatives for regular expressions.

Definition. A regular expression r is **nullable** if the language it defines contains the empty string, that is, if $\epsilon \in \mathcal{L}[[r]]$.

We also need the following function:

$$\delta(r) = \begin{cases} \epsilon & \text{if } \epsilon \in \mathcal{L}[[r]] \\ \emptyset & \text{if } \epsilon \notin \mathcal{L}[[r]] \end{cases}$$

The δ function takes REs to REs (recall that the empty set is a symbol set, which is an RE). Intuitively, δ collapses an RE to the “smallest” RE with the same nullability.

The following function, due to Brzozowski, gives the derivative of a regular expression with respect to a symbol a .

$$\begin{aligned} D_a(\epsilon) &= \emptyset \\ D_a(\mathcal{S}) &= \begin{cases} \epsilon & \text{if } a \in \mathcal{S} \\ \emptyset & \text{if } a \notin \mathcal{S} \end{cases} \\ D_a(r \cdot s) &= D_a(r) \cdot s \mid \delta(r) \cdot D_a(s) \\ D_a(r^*) &= D_a(r) \cdot r^* \\ D_a(r \mid s) &= D_a(r) \mid D_a(s) \\ D_a(r \& s) &= D_a(r) \& D_a(s) \\ D_a(\neg r) &= \neg D_a(r) \end{aligned}$$

We can take the derivative of an RE with respect to a sequence of symbols in a straightforward way:

$$\begin{aligned} D_\epsilon(r) &= r \\ D_{ua}(r) &= D_a(D_u(r)) \end{aligned}$$

Intuitively, the derivative of an RE with respect to a symbol a yields a new RE after matching a . The following two theorems, again due to Brzozowski, make this precise.

Theorem. The derivative $D_s(r)$ of any regular expression r with respect to any sequence u is a regular expression.

Theorem. A sequence u is contained in $\mathcal{L}[[r]]$ if and only if $\mathcal{L}[[D_u(r)]]$ is nullable.

Derivatives provide an easy method of DFA construction. Suppose we want to build a DFA that recognizes r . We can think of each state of the DFA as a regular expression. We start with a state Q_0 that represents r . We then take the derivative of r with respect to each symbol of the alphabet and create a new state each time a new derivative is found, adding each new state to the work list. We pop a state from the work list and repeat, until the work list is empty. There will be a transition from Q_j to Q_k if and only if (identifying states and their REs) $D_a(Q_j) = Q_k$ for some symbol a ; the transition will

be labeled with the set of all such a . Finally, any state that represents a nullable RE is an accepting state. The correctness of the recognizer is a direct consequence of the above theorems.

The sketch glosses over several important details. First, what notion of equality do we intend for the equation $D_a(Q_j) = Q_k$? Ideally, we would identify as a single state all those REs which admit the same language, so that $D_a(Q_j) = Q_k$ if and only if $\mathcal{L}[D_a(Q_j)] = \mathcal{L}[Q_k]$. This is expensive to compute, so Brzozowski introduced the notion of RE similarity, an equivalence on REs which is easy to compute but still guarantees that the DFA is finite.

Let \approx denote the least equivalence relation on REs such that

$$\begin{aligned} r \mid r &\approx r \\ r \mid s &\approx s \mid r \\ (r \mid s) \mid t &\approx r \mid (s \mid t) \end{aligned}$$

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

Theorem. Every regular expression has only a finite number of dissimilar derivatives.

Hence, DFA construction is guaranteed to succeed if new states are only created when no existing state is similar to a given derivative. In fact, we want to do much better than this to avoid blowup in DFA size.

Remark. In a practical implementation of DFA construction using derivatives, it is crucial to aggressively identify when a derivative admits the same language as an existing state (RE) in the DFA. The cost of this identification must be balanced against the number of duplicate states avoided.

In ML-Flex, we accomplish this by canonicalizing all input and derived REs. The canonicalization is described in detail in section 3.1.

2.3 Factorings

Another problem with DFA construction is the size of the unicode alphabet: taking the derivative with respect to each unicode symbol is not feasible. But to construct the DFA, we have to examine every possible derivative of a given RE. We must try to conservatively estimate what sets of symbols will yield the same derivative for an RE. Here we break from Brzozowski's work and introduce new terminology and an algorithm to make derivatives more amenable to large alphabets.

Let \sim_r be the relation defined as follows. For a regular expression r and symbols a, b , $a \sim_r b$ if and only if $D_a(r) = D_b(r)$.

Definition. The **derivative classes** of r are the the equivalence classes Σ/\sim_r .

Ultimately, the outedges for a DFA state and the derivative classes of the RE for that state are in one-to-one correspondence.¹ Hence, we must eventually determine all the

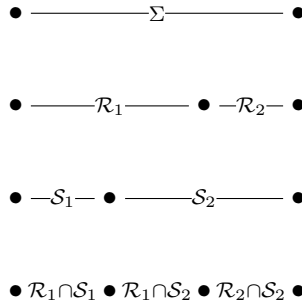
¹This is not quite true: we usually drop error transitions, that is, transitions going to the RE \emptyset .

derivative classes for an RE in order to construct the DFA. To avoid testing the entire alphabet a symbol at a time, we introduce an algorithm which (over)partitions Σ , so that each partition is a subset of a derivative class. We can then take the derivative with respect to a representative from each partition, and determine which partitions actually belong to the same derivative class.

Definition. Let r be an RE. A **factoring** of Σ under r is a partitioning of Σ such that each partition is a subset of a derivative class for r .

To be clear: we are factoring the *alphabet* into partitions, but the factoring is guided by (*under*) a regular expression. A factoring under a given RE is not unique. The derivative classes for an RE are one possible factoring (with a minimal number of partitions) while the set of all singleton sets of symbols is another factoring (with a maximal number of partitions). We will present a simple recursive factoring algorithm and prove its correctness, but first, an example.

Suppose we have two regular expressions r and s yielding factorings $\{\mathcal{R}_1, \mathcal{R}_2\}$ and $\{\mathcal{S}_1, \mathcal{S}_2\}$ respectively. Let $t = r \mid s$. The derivative of t with respect to some symbol a is $D_a(t) = D_a(r) \mid D_a(s)$. Hence, if $D_a(r) = D_b(r)$ and $D_a(s) = D_b(s)$ for some symbols a, b , then $D_a(t) = D_b(t)$ and so $a \sim_t b$. We can use this to give a factoring under t . The relationship between the factorings under r, s and t can be visualized as follows:



This small example captures the essential idea of the algorithm. To give a factoring under an RE, we recursively find factorings under its components and “compress” those factorings into a single new factoring that respects them. The factorings are being compressed (flattened) in the sense that the boundaries of one factoring are forced onto another, causing some partitions to split. The algorithm we present is in two stages: first, a factoring function recursively collects factorings under an RE; then, a compress function compresses them all onto Σ to produce a single factoring for an RE. We now make this precise.

The *factoring* function F takes a regular expression and gives a factoring of Σ under

that RE. It is defined recursively as follows:

$$\begin{aligned}
F(\epsilon) &= \emptyset \\
F(\mathcal{S}) &= \{\mathcal{S}\} \\
F(r \cdot s) &= \begin{cases} F(r) & \epsilon \notin \mathcal{L}[[r]] \\ F(r) \cup F(s) & \text{otherwise} \end{cases} \\
F(r \mid s) &= F(r) \cup F(s) \\
F(r \& s) &= F(r) \cup F(s) \\
F(r^*) &= F(r) \\
F(\neg r) &= F(r)
\end{aligned}$$

The *compress* function $C : \mathcal{P}(\Sigma) \longrightarrow \mathcal{P}(\Sigma)$ takes a set of subsets of the alphabet and produces the smallest partitioning of Σ that respects them. In particular, if

$$C(\{\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_m\}) = \{\mathcal{S}'_1, \mathcal{S}'_2, \dots, \mathcal{S}'_n\}$$

then we have that $\{\mathcal{S}'_1, \mathcal{S}'_2, \dots, \mathcal{S}'_n\}$ is a partitioning of Σ such that for each \mathcal{S}'_i and \mathcal{S}_k either $\mathcal{S}'_i \subseteq \mathcal{S}_k$ or $\mathcal{S}'_i \cap \mathcal{S}_k = \emptyset$.

Theorem. *Let r be an RE. Then $C(F(r))$ is a factoring of Σ under r .*

Proof: by induction on the structure of r . We use a to denote an arbitrary symbol.

Case ϵ : we have $D_a(\epsilon) = \emptyset$ for all $a \in \Sigma$, so $\Sigma/\sim_\epsilon = \{\Sigma\}$. We have $C(F(\epsilon)) = C(\{\emptyset\}) = \{\Sigma\}$.

Case \mathcal{S} : we have $D_a(\mathcal{S}) = \epsilon$ if $a \in \mathcal{S}$ and $D_a(\mathcal{S}) = \emptyset$ otherwise. Thus the derivative classes are \mathcal{S} and $\Sigma \setminus \mathcal{S}$, which are exactly the sets produced by $C(F(\mathcal{S})) = C(\{\mathcal{S}\})$.

Case $s \cdot t$ and $\epsilon \notin \mathcal{L}[[s]]$: here $D_a(s \cdot t) = D_a(s) \cdot t$. Because t is fixed as a varies, the derivative classes are just the derivative classes of s . Since $F(s \cdot t) = F(s)$ the result holds by the induction hypothesis on s .

Case $s \cdot t$ and $\epsilon \in \mathcal{L}[[s]]$: here $D_a(s \cdot t) = D_a(s) \cdot t \mid \epsilon \cdot D_a(t)$. Let $b, c \in \Sigma$ such that $b \sim_s c$ and $b \sim_t c$. Then $b \sim_{s \cdot t} c$. The result follows from this fact and the inductive hypothesis applied to s and t .

The other cases are similar.

3 Code

3.1 RegExp

In ML-Flex, REs are captured by the abstract type `RegExp.re`. Introduction is provided by various “smart constructors” (`mkSym`, `mkClosure`, ...), and elimination is provided by the derivatives algorithm. The signature for the `RegExp` module is shown below:

```
signature REG_EXP =
  sig

    structure Sym : INTERVAL_DOMAIN
    structure SymSet : INTERVAL_SET

    type symbol
    type sym_set
    type re

    val any      : re (* wildcard *)
    val none     : re (* EMPTY language *)
    val epsilon  : re (* the nil character (of length 0) *)

    val mkSym    : symbol -> re
    val mkSymSet : sym_set -> re

    val mkOr     : re * re -> re
    val mkAnd    : re * re -> re
    val mkXor    : re * re -> re
    val mkNot    : re -> re
    val mkConcat : re * re -> re
    val mkClosure : re -> re
    val mkOpt    : re -> re
    val mkRep    : re * int * int -> re
    val mkAtLeast : re * int -> re

    val isNone   : re -> bool
    val nullable : re -> bool
    val derivative : symbol -> re -> re
    val derivatives : re Vector.vector ->
      ((re Vector.vector) * sym_set) list

    val symToString : symbol -> string
    val toString    : re -> string
    val compare     : re * re -> order

  end
```

The included structure `SymSet` provides symbol interval sets, which are ideal when working with dense sets such as unicode character classes. Interval set operations (`union`, `complement`, ...) are used extensively; documentation for the interval set library is available with the SML/NJ distribution.

Recall that, in using RE derivatives for DFA construction, it is important to aggressively identify when two REs generate the same language so that they may be merged to a single state in the automaton. `RegExp` canonicalizes REs, which is why its `re` type is abstract. Canonicalization is performed using a lexicographic ordering on REs given by the `compare` function. The comparison is lexicographic in the sense that it first examines the top-level operation of the two REs, and only does more comparisons if that operation is the same. We represent REs as follows:

```
datatype re
  = Epsilon
  | Any
  | None
  | SymSet of sym_set
  | Concat of re list
  | Closure of re
  | Op of (rator * re list)
  | Not of re
and rator = OR | AND | XOR
```

For the `Op` constructor, which is used for three different commutative operations, the sub-REs are to be listed in canonical order. The `compare` function itself will expect that this is the case, and since only smart constructors can be used to construct REs, the invariant will always hold.

Smart constructors do additional canonicalization beyond ordering. For example, `mkNot (mkNot (none))` will be canonicalized to the same representation as `none`. Several similar RE equalities are detected and used. Also, the smart constructors attempt to push the RE representation as much toward symbol sets as possible, replacing boolean operations at the RE level with a single resulting symbol set when possible. For example, suppose that `AM` held the symbol set for $[A - M]$, `HZ` the set for $[H - Z]$, and `AZ` the set for $[A - Z]$. Then we would have

```
compare (mkSymSet AZ, mkOr (mkSymSet AM, mkSymSet HZ)) = EQUAL
```

The hope is that the cumulative effect of such canonicalization will keep the generated DFA close to minimal size.

The `derivative` function constructs a canonicalized derivative for a given RE with respect to a given symbol; it is a transcription of the algorithm described in section 2.2.

In general, we will be interested in the derivatives of a vector of REs (representing the rules in a lexer specification) with respect to *every* symbol in the alphabet. As section 2.3 explains, the unicode alphabet is too large for us to literally test the derivative at each symbol. The `derivatives` function will take a vector of REs and return a list of `re Vector.vector * sym_set pairs`. In fact, `derivatives` is just an implementation of

```

type action = string
type rule_spec = AtomSet.set option * RegExp.re
type rule = rule_spec * action

datatype spec = Spec of {
  decls : string,
  conf : config,
  rules : rule list
}

```

Figure 2: A fragment of LexSpec

the factor and compress algorithm given in 2.3. Each pair in the result list represents an arrow transitioning out of the current state (which corresponds to the input RE vector) to a new state (the output RE vector) on a given set of symbols (the output symbol set). Thus the labor of DFA construction is split between the RegExp module, which computes derivatives (and hence transitions) and LexGen, which actually constructs the DFA graph.

3.2 LexSpec and LexOutputSpec

Before describing DFA generation, we briefly discuss the relevant input and output data structures. The LexSpec module has data constructors and functions relevant to *e.g.* the ML-Lex input specification format, as shown in Figure 2

Actions, at least in the present implementation, are just raw strings. A rule consists of a rule specification and an associated action. The optional atom set included with a rule spec represents the start states to which that rule applies (with NONE meaning, strangely enough, all start states). With these definitions, a lexer specification is just a list of rules, some declarations (a raw string containing code) and some miscellaneous configuration data.

As Figure 3 illustrates, the output of DFA generation is *not* a DFA, but a collection of DFAs along with a vector of actions. Each machine represents a start state for the lexer; that is, each start state has its own automaton. However, since start states may use the same actions, we separate out the actions into a vector so that they are only emitted once.

A machine includes a label, which is just the name of the start state, as well as the rules relevant to that machine, which are paired with an index to the associated action in the action vector. The DFA itself is a list of states, with the head of the list being q_0 . A state, in turn, is labeled by a vector of REs (with the same length as the rules vector in the machine). Since a given DFA state may be an accepting state for more than one rule, we store a *list* of rule indices for the final flag. On a match, the action for the lowest-index rule is executed first, but that action may use REJECT(), in which case it may be necessary to jump to the action for the next rule index given in the final list.

```

datatype dfa_state
  = State of {
    id : int,
    label : RegExp.re Vector.vector,
    final : int list,
    next : (RegExp.sym_set * dfa_state) list ref
  }

datatype machine = Machine of {
  label : string,
  rules : (RegExp.re * int) vector,
  states : dfa_state list
}

type action = string

datatype spec = Spec of {
  actions : action vector,
  machines : machine list,
  ... (* configuration data *)
}

```

Figure 3: A fragment of LexOutputSpec

3.3 LexGen

LexGen is a very simple module with a single accessible function:

```
gen : LexSpec.spec -> LexOutputSpec.spec
```

The first task of the `gen` function is to collate the actions and start states. The actions from the input spec are separated from their rules into a vector. Afterwards, `gen` iterates over the specified start states, collecting the rules for each start state and using the `mkDFA` function to generate a machine for each.

`mkDFA` performs a straightforward imperative DFA construction, using the `derivatives` function from `RegExp` to compute the transitions out of each node. A map of RE vectors to DFA nodes is maintained (using `Vector.collate RegExp.compare` for ordering); this map is used to recognize when a new out-edge is going to an existing DFA node. Finally, any transition to a vector of REs which all generate the empty language is an error transition (that is, the derivatives for the given transition symbol set all indicate a non-match).

3.4 Front ends

At the moment, only one front end is available: the ML-Lex specification format. Eventually ML-Flex will have its own format.

The ML-Lex front end is fairly straightforward: it uses ML-Yacc to do the parsing, and can use either ML-Lex or ML-Flex to do the lexing. The distribution includes a lexer that was generated from ML-Flex to serve as a bootstrap.

3.5 Back ends

One exciting aspect of ML-Flex is the ability to easily add back ends. The following back ends are currently included in the distribution.

SML control-flow lexer generation The most important back end is code generation for SML. At present, there is a single code generation strategy: build a lexer using control-flow (*i.e.* if statements and tail-calls) to match the input. If the control-flow strategy turns out to be inadequate, it would be fairly easy to add a code generator for table-based lexing.

Code generation is performed by expanding stubs in a template SML file. The template contains code for dealing with streams and setting up the lexer in the appropriate way. Input is read from functional strings, which allow for the arbitrary lookahead needed for maximal-munch. These streams also track the current file position and line number.

A helper module, `ML`, contains a representation for a good portion of the Standard ML expression language, and support for pretty-printing such expressions.

The generated code includes user actions and the DFA for each start state. Within each start state there is one *state function* per DFA node. Each state function will examine the next character of input and perform a series of if-tests to determine the correct transition. The if-tests perform a hard-coded binary search over the symbol set intervals for the transitions.

The lexer stores a reference cell for the current functional stream. Each time the lexer accepts a token, the cell is updated. `yytext` is generated (only when needed) by “subtracting” the new stream from the one stored in the reference cell before updating it. Subtraction simply rescans the appropriate number of characters from the stored stream, which should still have the string in its buffer.

One subtlety is handling calls to `REJECT()`. Since such calls are fairly rare, we want to avoid tracking the information `REJECT` needs if it won’t be used. `REJECT` information consists of (in essence) a list of all the previous matches. A previous match could occur on the same DFA state if multiple REs matched the input at that state; otherwise, the previous match is a prefix of the `REJECT`ed match. Whenever a transition is made, the appropriate previous match information is passed up to the next state. However, if the current state is an accepting state that does not use `REJECT`, then the history is *truncated* at the `REJECT`-free point, since it could never be used. This determination is made statically and is hard-coded into the generated SML lexer.

Graphviz The Graphviz toolkit provides easy graph visualization using a very simple graph description format. We utilize the DOT format. The Graphviz backend will write one DOT file for each start state, showing all nodes for that start state’s DFA, along with labeled transitions. See www.graphviz.org for details on the DOT format.

Text dump If Graphviz is not available or a more detailed summary is desired, the `DumpOutput` module can be used to print to standard out every DFA state, labeled by its RE vector, along with out-edges, for every start state in the lexer.

Interactive matching Finally, a simple interactive back end is available. Interactive matching allows a user to enter arbitrary strings and find out (1) if they matched and (2) what RE matched them. The code for interactive matching is quite simple.

3.6 Main

The `Main` module is the driver for ML-Flex: it is responsible for processing command-line arguments and hooking up the appropriate front and back ends. Since it is very simple, it gives a nice overview of the system and is a good place to look first in trying to understand the code.

References

[Brz64] Brzozowski, J. A. Derivatives of regular expressions. *J. ACM*, 11(4), 1964, pp. 481–494.