Converting Regular Expressions to Non-Deterministic Finite Automata

Originally:
comp.compilers and comp.theory
1992 May 17
Modified 1996 May 17; 2005 December 22

This is a demo program developed and released originally in 1992 May 17 whose purpose is to illustrate an efficient algorithm for converting Regular Expressions to Non-Deterministic Finite Automata. The algorithm is described below.

1. Input and Output

1.1. The Underlying Algebra

This program is run from the command line and accepts input from standard input or by file redirection from a file. It will accept a regular expression listed in more or less standard syntax described below, which also contains a facility for defining subexpressions:

0	to denote the empty set
1	to denote the empty string
x	input symbol. Can be a C identifier or string literal.
$[A]$, $A^?$	$= 1 \mid A$.
A^+	$= AA^*$.
A^*	**iteration** ($A^* = 1 \mid A \mid AA \mid AAA \mid \ldots$)
$A \mid B$	**alteration** (the union of A and B)
AB	**concatenation**
$x = A, B$	**substitution** ($= B[x/A]$)

The operations form an algebra that has a product and identity satisfying the defining relations of what is called a *monoid*:

$$(xy)z = x(yz), \quad x1 = x = 1x.$$

It is partially ordered in such a way that every finite subset A has a least upper bound $\sup A$ with

$$0 = \sup \varnothing, \quad x \mid y = \sup\{x, y\}, \quad 0 \mid x_1 \mid x_2 \mid \ldots \mid x_n = \sup\{x_1, x_2, \ldots, x_n\}.$$

The operation $x \mid y$ may equally well be written as a sum $x + y$, in which case its most distinguishing property which sets it apart from the more familiar addition operation is the identity $x + x = x$ known as *idempotency*. Such algebras are, thus, termed *idempotent semirings*, or (since they build atop the monoid): *dioids*.[1]

Finally, the dioid has the property that every finite linear system of inequalities, written here in matrix form,

$$X \geq AX + B$$

has a least fixed point solution given (again, in matrix form) by $X = A^*B$ with the matrix for given recursively in terms of smaller submatrices by the decomposition:

$$\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)^* = \left(\begin{array}{c|c} N & NBD^* \\ \hline D^*CN & D^* \mid D^*CNBD^* \end{array}\right), \quad N = \left(A \mid BD^*C\right)^*.$$

That defines what's known as a *Kleene algebra*. It is a basic result of the respective algebraic formalisms that if one starts with an arbitrary set X and builds up terms, subjecting them only to the axioms of the respective type of algebra, then one obtains as a result the *free monoid* X^* of all finite words taken from the alphabet X, the *free dioid* $\mathrm{F}\left(X^*\right)$ of all finite subsets of words taken from X^* (i.e. all *finite languages)*, and the *free Kleene algebra* $\mathrm{R}\left(X^*\right)$ of all rational subsets of words taken from X^* – i.e., what we also call the *Regular Expressions* over the alphabet X.

[1] Our preference for using $A \mid B$ to denote the addition/least upper bound operator rather than $A + B$ is to avoid a clash of notation since we're already using A^+ to denote the "1-or-more" iteration operator.

Going by the suggestion in the initial summary, it is also possible to add the condition

$$x^* = \sup_{n\geq 0} x^n = 1 | x | xx | xxx | ...$$

This property is called **-continuity*, but does not follow for general Kleene algebras. Expanding on the previous remark, it is also a basic result that if one take an already-existing monoid M and adds in the Kleene star, subjecting them only to the Kleene algebra identities and *-continuity, then the result will be $\mathrm{R}M$, the *rational subsets* of M. A similar result cannot be obtained for Kleene algebras in general without *-continuity.

The most import case of Kleene algebras other than the regular expressions are the algebras $\mathrm{R}\left(X^* \times Y^*\right)$ of *rational transductions* obtained by finite state machines with input alphabets X and output alphabets Y. At present, the software cannot directly deal with rational transductions, nor generalizations $\mathrm{R}\left({X_0}^* \times {X_1}^* \times \ldots \times {X_{n-1}}^*\right)$ to $n \geq 3$ channels.

1.2. The Program Operation

Assignments are processed sequentially, so a variable that appears on both sides of an equation does *not* make a recursive equation (otherwise, we'd be talking about Context Free grammars here).

For example, these are valid regular expressions:

$$\left(\mathbf{a}\left[\mathbf{b}^+\mathbf{a}^*\right]\right)^+ | \mathbf{c}^*\mathbf{ab}, \quad \mathbf{a}^*\left(\mathbf{ba}^*\right)^*.$$

The output is a nondeterministic finite automaton listed in equational form. For example, the NFA corresponding the regular expressions above are, respectively:

$$\begin{array}{ll} \left(\mathbf{a}\left[\mathbf{b}^+\mathbf{a}^*\right]\right)^+ | \mathbf{c}^*\mathbf{ab} & \mathbf{a}^*\left(\mathbf{ba}^*\right)^* \\ Q_0 = \mathbf{a}Q_3 | \mathbf{a}Q_2 | \mathbf{c}Q_1, & Q_0 = \mathbf{a}Q_0 | 1 | \mathbf{b}Q_0. \\ Q_1 = \mathbf{a}Q_2 | \mathbf{c}Q_1, & \\ Q_2 = \mathbf{b}Q_4, & \\ Q_3 = \mathbf{b}Q_5 | 1 | \mathbf{a}Q_3, & \\ Q_4 = 1, & \\ Q_5 = \mathbf{b}Q_5 | \mathbf{a}Q_6 | 1 | \mathbf{a}Q_1, & \\ Q_6 = \mathbf{a}Q_6 | 1 | \mathbf{a}Q_3, & \end{array}$$

The left-hand side represents the state, and the right hand side the sum of all the alterands. A 1 indicates that this state is an accepting state. For the first regular expression, Q_3 and Q_5 are both accepting states (as is Q_6). A symbol followed by a state name denotes an arc. For example, state Q_0 has arc labeled **a** pointing to state Q_3, and an arc labeled **c** pointing to Q_2. The alterands are separated from each other by a "|".

State Q_0 is always the starting state.

2. Diagnostics

Syntax errors, where they occur will be listed in the following form:

[Line Number] Descriptive error message.

All error messages are directed to the standard error file, not to standard output file.

3. The Algorithm

3.1. Reduction as Applied Algebra

One of the main motivations behind doing this was to get rid of the Rube Goldberg engineering mentality that seems to pervade these and other areas in computer science (which, among other unfortunate effects, leads to such lumbering monstrosities as the "regex" package bundled with a typical computer or OS), and bring computer science back to mathematics (and physics) where it belongs. Computer Science is, after all, a branch of Mathematics and Logic, not engineering (thus, for instance, it is classified as QA by the Library of Congress Classification with Mathematics). The

resulting simplification is why this direction of pursuit should be innate to anyone who pursues this with a clear mind, never mind the elegance of the results achieved.[2]

A set-valued function $E \mapsto \langle E \rangle$ is defined over regular expressions E recursively as follows

$$\langle [A] \rangle = \langle A^? \rangle = \langle 1 \mid A \rangle, \quad \langle A^+ \rangle = \langle AA^* \rangle, \quad \langle A^* \rangle = \langle 1 \mid A^+ \rangle,$$
$$\langle [A]C \rangle = \langle A^?C \rangle = \langle C \mid AC \rangle, \quad \langle A^+C \rangle = \langle A(A^*C) \rangle, \quad \langle A^*C \rangle = \langle C \mid A^+C \rangle,$$
$$\langle 0C \rangle = \langle 0 \rangle = \varnothing, \quad \langle 1C \rangle = \langle C \rangle, \quad \langle (A \mid B)C \rangle = \langle AC \mid BC \rangle, \quad \langle (AB)C \rangle = \langle A(BC) \rangle,$$
$$\langle 1 \rangle = \{1\}, \quad \langle x \rangle = \{x1\}, \quad \langle xQ \rangle = \{xQ\}, \quad \langle A \mid B \rangle = \langle A \rangle \cup \langle B \rangle.$$

The definition may cycle. Applying these criteria to a given regular expression will result in a fixed-point system consisting of a finite set of set-theoretic equations. The minimal solution to this system is calculated and written out.

By induction, the following can then be proven for the mapping $E \mapsto \langle E \rangle$:

(A) The only terms in $\langle E \rangle$ are those of the form 1 or xQ for some symbolic name x, and expression Q.

(B) The size of $\langle E \rangle$ is linear in number of operations, variables and constants in E.

(C) E = sum of all terms in $\langle E \rangle$.

The outcome of the process may, in fact, is given by $\langle A \rangle = A_0 \cup dA$ where

A	0	1	x	$A^?$	A^*	A^+	$A \mid B$	AB
dA	$\varnothing$	$\varnothing$	$\{x1\}$	dA	$dA \cdot \{A^*\}$	$dA \cdot \{A^*\}$	$dA \cup dB$	$dA \cdot \{B\} \cup A_0 dB$
A_0	$\varnothing$	$\{1\}$	$\varnothing$	$\{1\}$	$\{1\}$	A_0	$A_0 \cup B_0$	$A_0 B_0$

The subexpression Q in each term xQ is then considered to be a state in the NFA. Each newly generated state is likewise reduced to normal form in the same way, and this in turn may generate more new states. When all the states that were generated from the top-level expression have been fully reduced, the result is an NFA representing the original expression. This process will always halt in a time and space that is *linear* with respect to the size of the expression's parse tree with a coefficient ≤ 2.

For example, take this expression $E = \mathbf{a}^*(\mathbf{b}\mathbf{a}^*)^*$. For convenience, let

$$x_0 = \mathbf{a}^*, \quad x_1 = x_2{}^*, \quad x_2 = \mathbf{b}x_0.$$

Then E reduces as follows

$$\langle E \rangle = \langle x_0 x_1 \rangle = \langle \mathbf{a}^* x_1 \rangle \to \langle x_1 \mid \mathbf{a}^+ x_1 \rangle = \langle x_1 \rangle \cup \langle \mathbf{a}^+ x_1 \rangle,$$
$$\langle x_1 \rangle = \langle x_2{}^* \rangle \to \langle 1 \mid x_2 x_1 \rangle = \langle 1 \rangle \cup \langle x_2 x_1 \rangle,$$
$$\langle x_2 x_1 \rangle = \langle (\mathbf{b}x_0) x_1 \rangle \to \langle \mathbf{b}(x_0 x_1) \rangle = \langle \mathbf{b}E \rangle = \{\mathbf{b}E\},$$
$$\langle \mathbf{a}^+ x_1 \rangle \to \langle \mathbf{a}(\mathbf{a}^+ x_1) \rangle = \langle \mathbf{a}E \rangle = \{\mathbf{a}E\}.$$

Therefore, $\langle x_1 \rangle = \{1, \mathbf{b}E\}$ and $\langle E \rangle = \langle x_1 \rangle \cup \langle \mathbf{a}^+ x_1 \rangle = \{1, \mathbf{b}E, \mathbf{a}E\}$.

The software I wrote takes these exact steps to derive an NFA from the regular expression E. And it just so happens that for this example, E is the only "state" in the NFA. You may want to compare this with the one described in Berry and Sethi ("From Regular Expressions to Deterministic Automata" Theoretical Computer Science 1986). Their example:

---

[2] It is also intended as a shot across the bow directed against governments that purport to classify the "innovations" of computer science as "devices" or "patentable" or "ownable" objects, rather than as mathematics. The "shot across the bow" is also meant to serve advanced notice that no such edict or law will be recognized, nor shall anything premised on it be. Not even the tacit acceptance of this state of affairs known as a "copyleft".

$$(\mathbf{a}_1\mathbf{b}_2 \mid \mathbf{b}_3)^*\mathbf{b}_4\mathbf{a}_5$$

with the my algorithm will yield the automaton below

$$Q_0 = \mathbf{b}_4 Q_2 \mid \mathbf{a}_1 Q_1 \mid \mathbf{b}_3 Q_0, \quad Q_1 = \mathbf{b}_2 Q_0, \quad Q_2 = \mathbf{a}_5 Q_3, \quad Q_3 = 1.$$

3.2. Toward a Generalization to Rational Transductions

As an aside, it is possible to generalize this so as to be able to deal with rational transductions, where an output alphabet Y has been introduced. In that case we will generalize from being binary $A_0 \in \{0,1\}$ to entire subsets $A_0 \in \mathrm{R}\,Y^*$ corresponding to alls *empty actions* on state A. The resulting additions and modifications are:

y	$A^?$	A^*	A^+
$\varnothing$	dA	$(A_0)^*\, dA \cdot \{A^*\}$	$(A_0)^+\, dA \cdot \{A^*\}$
$\{y\}$	$(A_0)^?$	$(A_0)^*$	$(A_0)^+$

Each element of dA then has a form $\alpha x B$ consisting of a regular expression $\alpha \in \mathrm{R}\,Y^*$ representing the entire set of actions that may take place on an input symbol x in making the transition to state B. In other words, we have an *idempotent power series* on the set X^* with coefficients taken from the idempotent algebra $\mathrm{R}\,Y^*$. Such a concept, as far as I'm aware, has not been entertained in the literature before. Indeed, every rational family $\mathrm{R}\,(M \times N)$ reduces to an idempotent power series over the general monoid M with coefficients taken from $\mathrm{R}\,N$. In contrast, the literature generally only deals with power series restricted to free monoids X^*.

4. Implementation

This is a short summary detailing some of the internal workings of the program.

4.1. Parsing

The parser used was generated by an unpublished algorithm based, in part, on the algorithm presented above, but greatly expanding it. The parser was derived from the syntax:

$$E \to \mathbf{c} \mid \mathbf{x} \mid (E) \mid [E] \mid E\mathbf{p} \mid EE \mid E\mathbf{i}E \mid \mathbf{x} = E, E,$$
$$E$$

where **c** denotes a constant (i.e. **0** or **1**), **x** a terminal, **p** a postfix operator (i.e. $*$, $+$ or $?$) and **i** an infix operator (which, here, is just $|$). Natural precedence rules are used to resolve ambiguities.

The individual phrase structure rules may be labeled like so:

$$\alpha : E \to \mathbf{c}, \quad \beta : E \to \mathbf{x}, \quad \gamma : E \to (E),$$
$$\delta : E \to (E), \quad \varepsilon : E \to E\mathbf{p}, \quad \zeta : E \to EE,$$
$$\eta : E \to E\mathbf{i}E, \quad \theta : E \to \mathbf{x} = E, E, \quad \omega : \to E,$$

where the last rule is the "accept" action. Associated with the grammar is a *canonical* top-down syntax directed transduction, as well as a canonical bottom-up syntax directed transduction (SSDT). Each rule is identified as an action. For instance, for the rules α and ω: for top-down parsers the action $\alpha = \{E \Rightarrow \mathbf{c}\}$ is to expand E into **c** and $\omega = \{\Rightarrow E\}$ is to start the process with E; while for bottom-up parsers, the action $\alpha = \{E \Leftarrow \mathbf{c}\}$ is to reduce **c** into E and $\omega = \{\Leftarrow E\}$ is to finish the process by accepting E. The corresponding SSDT's may be written as follows:

Top-Down Canonical SSDT

$$E \to \alpha\mathbf{c} \mid \beta\mathbf{x} \mid \gamma(E) \mid \delta[E] \mid \varepsilon E\mathbf{p} \mid \zeta EE \mid \eta E\mathbf{i}E \mid \theta\mathbf{x} = E, E, \omega E,$$

Bottom-Up Canonical SSDT

$$E \to \mathbf{c}\alpha \mid \mathbf{x}\beta \mid (E)\gamma \mid [E]\delta \mid E\mathbf{p}\varepsilon \mid EE\zeta \mid E\mathbf{i}E\eta \mid \mathbf{x} = E, E\theta, E\omega.$$

Each such syntax specifies a subset of $T \subseteq X^* \times Y^*$ called a *simple syntax directed translation* or *push down transduction*, with input alphabet X and output alphabet Y respectively given by:

X:					Y:		
c	**x**	(	)	[	α	β	γ
]	**p**	**i**	=	,	δ	ε	ζ
					η	θ	ω

The subset is just a context-free subset of $X^* \times Y^*$ – with the twist that the underlying monoid is no longer free but has relations $xy = yx$ for $x \in X$ and $y \in Y$, i.e. $X^* \times Y^* \cong (X \cup Y)^* / \{(xy = yx : x \in X, y \in Y)\}$. Nevertheless, it is still meaningful to speak of the family $\mathcal{C}M$ of context-free subsets of an *arbitrary* monoid M, so what we are describing above is a both a subset $T \in \mathrm{C}(X^* \times Y^*)$, as well as a means of specifying such subsets. In our case, this subset is given by

$$T = \bigcup_{w \in X^*} \{w\} \times T_w,$$

where $T_w \subseteq Y^*$ is the set of all parsing sequences associated with the word $w \in X^*$. The task of writing a parser is to find an SSDT that yields translate sets T_w for each $w \in X^*$ that are of size 0 or 1. If any T_w has size 2 or more, then the word w has two or more parses and the SSDT is *non-deterministic* for that word $w \in X^*$. Some other means then has to be employed to remove the non-determinacy … or otherwise to "split" the universe itself into two or more parallel branches to handle each one separately.

Since the methods I use originated long ago as a means to elaborate/simplify/optimize the unwieldy results that come out of a typical LR parser (i.e. bottom-up parsing), it is the canonical bottom-up SSDT that I use.

In algebraic terms, the grammar may be regarded as a system of fixed point inequalities over an algebra. Here, the system is given equivalently by

$$\begin{aligned} E &\geq \mathbf{c}\alpha + \mathbf{x}\beta + (E)\gamma + [E]\delta + E\mathbf{p}\varepsilon + EE\zeta + E\mathbf{i}E\eta + \mathbf{x} = E, E\theta \\ &= (\mathbf{c}\alpha + \mathbf{x}\beta + (E)\gamma + [E]\delta + \mathbf{x} = E, E\theta) + E(\mathbf{p}\varepsilon + E\zeta + \mathbf{i}E\eta), \\ E &\geq (\mathbf{c}\alpha + \mathbf{x}\beta + (E)\gamma + [E]\delta + \mathbf{x} = E, E\theta)(\mathbf{p}\varepsilon + E\zeta + \mathbf{i}E\eta)^*, \end{aligned}$$

From this, writing $E_Q = (\mathbf{p}\varepsilon + E\zeta + \mathbf{i}E\eta)^*$, we can directly read off the modified grammar

$$\begin{aligned} &E \to \mathbf{c}\alpha E_Q \mid \mathbf{x}\beta E_Q \mid (E)\gamma E_Q \mid [E]\delta E_Q \mid \mathbf{x} = E, EE_Q, \\ &E_Q \to 1 \mid \mathbf{p}\varepsilon E_Q \mid E\zeta E_Q \mid \mathbf{i}E\eta E_Q, \\ &E\omega. \end{aligned}$$

All of the left recursions have been removed.

The remaining recursions are removed by introducing a set of bracketing operators – this is an application of a variation of the *Chomsky-Schuetzenberger Theorem* recast entirely within the algebraic language cast by the framework. The algebraic form of the theorem asserts the following: if we expand a *-complete Kleene algebra by adding a set of operator symbols $\langle 0|, \ldots, \langle n|$ and matching pairs $|0\rangle, \ldots, |n\rangle$ and apply the identities

$$\langle i||j\rangle = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}, \quad \langle i|x = x\langle i|, \quad |j\rangle x = x|j\rangle,$$

for all $i, j = 0, \ldots, n$ and input symbols $x \in X$, then the resulting Kleene algebra is powerful enough to represent the least fixed point solutions of *all* finite fixed point systems. In the case of the rational subsets $\mathrm{R}M$, for an *arbitrary* monoid, that means the expanded algebra can represent all context-free subsets $\mathcal{C}M$. Thus, this provides an algebra and calculus for *Context Free Expressions.* Moreover, by virtue of the fact that this expansion applies to $\mathrm{C}(X^* \times Y^*)$ and not just to $\mathrm{C}X^*$, it means we have an algebra for parsers (i.e. SSDT's) – that is: *transductions*, not merely *languages*.

The conversion applied is the following. Associated with each non-terminal n are two states n_S and n_F. The start expression $E\omega$ becomes $\langle 0|E_S$ and $E_F \to |0\rangle\omega$. The context $\langle 0||0\rangle$ is thus associated with E at the top level. Each

rule of the form $q \to w$ is converted to $q_S \to wq_F$ where the word w contains only X 's and Y 's. Each rule of the form $q \to wq'\beta$, where β is an arbitary sequence, is converted to $q_S \to w\langle i|q'_S$, with the assignment of a new context $\langle i||i\rangle$ associated with the appearance of q' within this rule. Each instance of each rule of the form $q \to \alpha q' w q''\beta$ (and there may be several in a given rule) is converted to $q'_F \to |i\rangle w\langle j|q''_S$, where $\langle i||i\rangle$ was a context already assigned to q' and $\langle j||j\rangle$ a new context assigned to q''. Finally, each rule of the form $q \to \alpha q'' w$, is converted to $q''_F \to |j\rangle wq_F$, where $\langle j||j\rangle$ was a context previously assigned to q''. As a special case of the first form, since all the cyclic left recursions have been removed, then rules of the form $q \to q'\beta$ can just as well be written as $q_S \to [i_1,\ldots,i_m]q'_S$, where the *projection* $[i_1,\ldots,i_m] = |i_i\rangle\langle i_1||\ldots||i_m\rangle\langle i_m|$ is taken over all the contexts $i_1,\ldots,i_m$ that shall have come to be associated with q. This takes the place of both $\langle i|$ and $|i\rangle$ since the projection is equal to its own reversal.

This syntax results in a 2 state parser, with 1 ambiguous state resolved using an "action" function which maps

Lexical Item × Context → Action

where Action is either a "shift" on the lexical item or a "reduce" in the given context.

4.2. Expression Indexing

The result of the parse is a directed acyclic graph representing the parse tree with all the repeating subtrees merged. Each subexpression that is parsed is assigned an index and is stored in a hash table to facilitate the search for and elimination of duplicates.

The indexing function (i(E)) is inductively defined in terms of an indexing function (index(x)) on symbols as follows:

i(0) = f0, i(1) = f1
i(x) = f2(index(x))
i([E]) = f3(i(E))
i(E+) = f4(i(E))
i(E*) = f5(i(E))
i(E1 E2) = f6(i(E1) XOR i(E2))
i(E1 | E2) = f7(i(E1) XOR i(E2))

where the ranges of the functions f2, f3, f4, f5, f6, f7 and the values f0, f1 constitute a disjoint partition of the range of the index function i(E), and XOR denotes the exclusive or operation applied to the binary representations of the index values. For simplicity, f2, f3, f4, f5, f6 and f7 are all chosen to be linear functions.

The routine GetExp(...) is responsible for looking up expressions with the indicated structure, creating them if necessary. This function therefore has the following formats:

GetExp('0')	0
GetExp('1')	1
GetExp('x', X)	The value of X (if assigned to)
GetExp('x', X)	The literal X (if not assigned to)
GetExp('?', A)	[A]
GetExp('*', A)	A*
GetExp('+', A)	A+
GetExp('.', A, B)	A B
GetExp('|', A, B)	A | B

4.3. Generating the NFA

The routine

FormState(Q)

will apply the reduction algorithm described above to the expression Q until normal form is reached. The algorithm is also recursively applied to each of the states that are generated until all the states have been fully reduced.

4.4. Post-Processing

The NFA that is generated is usually very close to being a minimal finite state automaton. But in general, it is not deterministic, so that it will be necessary to apply the standard algorithms to generate a minimal DFA from this NFA, if you want to obtain an DFA. Since the rest of the algorithm is believed to be linear, if there's any exponential blow-up in the process of converting a regular expression to a DFA, it will occur in this step.

At the end, a stub routine WriteStates() will display the results of the conversion in the format described above. This routine is what you would replace by the conversion routines.

The set <E>, which represents the result of reducing the expression E, is effectively converted back into an expression in the WriteStates() routine.

5. Bugs

There are none that I know of.