**Converting Regular Expressions to Non-Deterministic Finite Automata**Originally:

comp.compilers and comp.theory

1992 May 17Modified 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:

|  |  |
| --- | --- |
|  | to denote the empty set |
|  | to denote the empty string |
|  | input symbol. Can be a C identifier or string literal. |
| , |  |
|  | . |
|  | **iteration** () |
|  | **alteration** (the union of  and ) |
|  | **concatenation** |
|  | **substitution** () |

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



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



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

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



has a least fixed point solution given (again, in matrix form) by  with the matrix for given recursively in terms of smaller submatrices by the decomposition:



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  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*  of all finite words taken from the alphabet , the *free dioid*  of all finite subsets of words taken from  (i.e. all *finite languages)*, and the *free Kleene algebra*  of all rational subsets of words taken from  – i.e., what we also call the *Regular Expressions* over the alphabet .

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



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  and adds in the Kleene star, subjecting them only to the Kleene algebra identities and \*-continuity, then the result will be , the *rational subsets* of . 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  of *rational transductions* obtained by finite state machines with input alphabets  and output alphabets . At present, the software cannot directly deal with rational transductions, nor generalizations  to  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:



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



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,  and  are both accepting states (as is ). A symbol followed by a state name denotes an arc. For example, state  has arc labeled  pointing to state , and an arc labeled  pointing to . The alterands are separated from each other by a “|”.

State  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]](#footnote-2)

A set-valued function  is defined over regular expressions  recursively as follows



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 :

1. The only terms in  are those of the form  or  for some symbolic name , and expression .
2. The size of  is linear in number of operations, variables and constants in .
3.  = sum of all terms in .

The outcome of the process may, in fact, is given by  where

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

The subexpression  in each term  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 . For convenience, let

.

Then  reduces as follows

,

,

,

.

Therefore,  and .

The software I wrote takes these exact steps to derive an NFA from the regular expression . And it just so happens that for this example,  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:



with the my algorithm will yield the automaton below



**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  has been introduced. In that case we will generalize from being binary  to entire subsets  corresponding to alls *empty actions* on state . The resulting additions and modifications are:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Each element of  then has a form  consisting of a regular expression  representing the entire set of actions that may take place on an input symbol  in making the transition to state . In other words, we have an *idempotent power series* on the set  with coefficients taken from the idempotent algebra . Such a concept, as far as I’m aware, has not been entertained in the literature before. Indeed, every rational family  reduces to an idempotent power series over the general monoid  with coefficients taken from . In contrast, the literature generally only deals with power series restricted to free monoids .

**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:



where  denotes a constant (i.e.  or ),  a terminal,  a postfix operator (i.e. ,  or ) and  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:



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  and : for top-down parsers the action  is to expand  into  and  is to start the process with ; while for bottom-up parsers, the action  is to reduce  into  and  is to finish the process by accepting . The corresponding SSDT’s may be written as follows:



Each such syntax specifies a subset of  called a *simple syntax directed translation* or *push down transduction*, with input alphabet  and output alphabet  respectively given by:



The subset is just a context-free subset of  – with the twist that the underlying monoid is no longer free but has relations  for  and , i.e. . Nevertheless, it is still meaningful to speak of the family  of context-free subsets of an *arbitrary* monoid , so what we are describing above is a both a subset , as well as a means of specifying such subsets. In our case, this subset is given by



where  is the set of all parsing sequences associated with the word . The task of writing a parser is to find an SSDT that yields translate sets  for each  that are of size 0 or 1. If any  has size 2 or more, then the word  has two or more parses and the SSDT is *non-deterministic* for that word . 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



From this, writing , we can directly read off the modified grammar



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  and matching pairs  and apply the identities



for all  and input symbols , 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 , for an *arbitrary* monoid, that means the expanded algebra can represent all context-free subsets . Thus, this provides an algebra and calculus for *Context Free Expressions.* Moreover, by virtue of the fact that this expansion applies to  and not just to , 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  are two states  and . The start expression  becomes  and . The context  is thus associated with  at the top level. Each rule of the form  is converted to  where the word  contains only ’s and ’s. Each rule of the form , where  is an arbitary sequence, is converted to , with the assignment of a new context  associated with the appearance of  within this rule. Each instance of each rule of the form  (and there may be several in a given rule) is converted to , where  was a context already assigned to  and  a new context assigned to . Finally, each rule of the form , is converted to , where  was a context previously assigned to . As a special case of the first form, since all the cyclic left recursions have been removed, then rules of the form  can just as well be written as , where the *projection*  is taken over all the contexts  that shall have come to be associated with . This takes the place of both  and  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.

1. Our preference for using  to denote the addition/least upper bound operator rather than  is to avoid a clash of notation since we’re already using  to denote the “1-or-more” iteration operator. [↑](#footnote-ref-1)
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”. [↑](#footnote-ref-2)