Converting Regular Expressions to Finite Automata

Originally:
1993 May 20
comp.compilers, comp.theory
Revised: 1993 October 21

1. Introduction

This is a demo program I wrote in 1992 that illustrates an algorithm that efficiently converts Regular Expressions to NFA's. It is based on a paper and pen algorithm I had been toying with the preceding week. An extra step was added the following year to convert NFA's into minimal DFA's using standard techniques. These steps were directly combined into an algorithm to directly produce DFA's from regular expressions using basically the Item Set construction familiar to those who work with LR(k) parsers.

In this version, the algorithm has been completely changed and extended to process a far more powerful regular expression notation. This notation includes the full set of boolean operators, and the Interleave Product operator, which basically models concurrency in finite automata.

The minimalization step was still kept separate and is subject to future improvement a la Hopcroft. The rest of the algorithm will be described below.

2. Input and Output

Input is read from standard input (or from a file by file redirection). It consists of series of comma-separated equations of the form:

Label = Regular Expression

followed by a regular expression. The expressions can consist of the following:

(0) 0 ------- to denote the empty set
(1) 1 ------- to denote the empty string
(2) x ------- any valid C identifier or string literal
(3) [A], A? ----- denotes the regular expression 1 | A.
(4) A+ ------ denotes the regular expression A A*.
(5) A* ------ ITERATION (the set 1 | A | A A | A A A | ...)
(6) A | B --- ALTERATION (the union of A and B)
(7) A B ----- CONCATENATION
(8) A & B --- INTERSECTION (the intersection of A and B)
(9) A - B --- DIFFERENCE (the set difference of A and B)
(10) A ^ B --- INTERLEAVE PRODUCT

The product A ^ B is defined as the set of all interleavings of expressions from A and B. Its finite automaton is the Cartesian Product of the finite automata for expressions A and B before conversion to a minimal DFA. The notation A x B would have been more appropriate, but that would confuse the demo program. The following illustrates the operation:

a ^ b = a b | b a

which has a finite automaton that looks like a diamond lattice,

a b ^ b a = (a ^ b) (a ^ b)

(two diamonds, one on top of the other),

a ^ b ^ c = a b c | a c b | b a c | b c a | c a b | c b a

(a cubic lattice). As an interesting observation, note that the sum:

1 | (a b)* | (a b)* ^ (a b)* | (a b)* ^ (a b)* ^ (a b)* | ...

will converge to the degree 1 Dyck language, defined recursively by the equation:

D[1] = 1 | a D[1] b | D[1] D[1]

or by:

D[1] = 1 | D[1] ^ (a b)*

This operation seems important enough that I've developed a notation for it (but not represented in this program):

A! = 1 | A | A^A | A^A^A | ...

(I'd like to say A-to-the-omega, but don't have an omega key). This is a Context Free Expression operator. The Dyck language D[1] is therefore equal to the expression (a b)*! (also equal to (a b)!). In fact, every Context Free Language can be represented using this operator, the other regular expression operators above, and the symbol-erasing operator:

x~E = { w1 w2 ... wn: w1 x w2 x ... x wn is in E, no x's in w1, w2 ... wn }

For instance, the CFL given by the equation

S = 1 | a S b

has the solution:

S = u1~v1~((a u1)* (v1 b)* & ((a | b)* ^ (u1 v1)!))

which is equivalent to the Context Free Expression:

<0| (a <1|)* (|1> b)* |0>

With the operators presented here, the expressions generated from recursive systems of equations can represent not only the Context Free Languages but a large superset, both of which go far beyond the processing ability of this program. However, you can generate successive approximations to this expression with input of the form:

S = 0,
S = 1 | S ^ (a b)*,
S = 1 | S ^ (a b)*,
S = 1 | S ^ (a b)*,
S = 1 | S ^ (a b)*,
S

These expressions generate a series of finite automata A0, A1, A2, ..., An, with the expression for A4 being given above. The resulting automaton "converges" to an infinite automaton which represents the Dyck Language D[1]. It has the form:

Q1 = 1 | a Q2,
Q2 = a Q3 | b Q1,
Q3 = a Q4 | b Q2,
Q4 = a Q5 | b Q3,
...

In the notation defined here, these are valid regular expressions:

(a [b+ a*])+ | c* a b
a* (b a*)*
(a | b)* - a* (b a*)*
a a (a | b)* & (a | b)* b b

The output is the minimal deterministic finite automaton listed in equational form. For example, the DFA corresponding the the regular expressions above are, respectively:

(a [b+ a*])+ | c* a b
Q1 = a Q2 | c Q3
Q2 = 1 | a Q2 | b Q2
Q3 = a Q4 | c Q3
Q4 = b Q5
Q5 = 1

a* (b a*)*
Q1 = 1 | a Q1 | b Q1

(a | b)* - a* (b a*)*
Q0 = 0

a a (a | b)* & (a | b)* b b
Q1 = a Q2
Q2 = a Q3
Q3 = a Q3 | b Q4
Q4 = a Q3 | b Q5
Q5 = 1 | a Q3 | b Q5

The left-hand side represents the state, and the right hand side the sum of all the terms. A 1 indicates that this state is an accepting state. For the first regular expression, Q2, and Q5 are both accepting states. A symbol followed by a state number denotes an arc. For example, state Q1 has arc labeled "a" pointing to state Q2, and an arc labeled "c" pointing to Q3. The terms are separated from each other by a "|".

State Q1 is always the starting state. State Q0 is reserved as the error state and is never shown, except when there are no other states (as in the third example).

3. 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.

4. The Algorithm

If E is a regular expression, then it will be reduced to the following Normal Form:

E = 1 | E1 | E2 | ...

where each term, E1, E2, ... has the form x E', where x is a symbol, and where each of the x's is distinct. While building up new expressions during parsing and during reduction, the following conversions will be constantly made:

(0) Operator zeroes are reduced: x 0 = 0 x = x & 0 = 0 & x = x ^ 0 = 0 ^ x = 0
(1) Operator identities are absorbed: x 1 = 1 x = x ^ 1 = 1 ^ x = x; x | 0 = 0 | x = x
(2) The terms in the commutative operators: A ^ B, A & B, A | B are listed in a consistent order
(3) The terms in the idempotent operators are merged: x & x = x | x = x
(4) The terms in the associative operators (A ^ B, A & B, A | B, A B) are grouped to the right (e.g. a | b | c | d = a | (b | (c | d))).

Given an expression, E, in order to reduce it to Normal Form make the following definitions:

\E = 1 if the empty string is in the set E,
0 else
i.e., \E = E & 1.

x\E = { w: x w is in the set E }

note that: x (x\E) = E & x (x1 | x2 | ... | xn)*

Then E can be written in the form:

E = \E | x1 (x1\E) | x2 (x2\E) | ... | xn (xn\E)

where x1, x2, ..., xn are the symbols in the input set. The following properties can then be derived from these definitions:

E	\E	x\E	Critical Subterms
0	0	0	-
1	1	0	-
x	0	1	-
y	0	0	-
[A]	1	x\A	A
A*	1	x\A A*	A
A+	\A	x\A A*	A
A B	\A \B	\A x\B | x\A B	B (if \A = 1) and A
A | B	\A | \B	x\A | x\B	A and B
A - B	\A - \B	x\A - x\B	A and B
A & B	\A \B	x\A & x\B	A and B
A ^ B	\A \B	(x\A^B) | (A^x\B)	A and B

Notice how closely that the x\E operator behaves like a differential operator with respect to the product operator. In the literature, it is often referred to as the derivative operator.

The algorithm consists of the following:

(1) All the \E's are calculated when the expression E is constructed.
(2) The operations x\E are calculated using the table.

The result of this calculation is that E is factored into Normal Form. Since E's calculation may rest on calculating the normal forms of some of its subexpressions, this calculation may need to be carried out recursively. Which subexpression need to be reduced and when are listed in the table above as the Critical Subterms.

It is absolutely critical that all the expressions encountered during the entire history of parsing and reduction be saved so that every single repeat occurrence of the expression can be looked up and eliminated. In this program, an expression hash table is used and the lookup is done each time a new expression is made and during each of the steps (0)-(4) above. What we're actually doing in this process is constructing "weak" equivalence classes based on syntatic similarity.

The automaton constructed is a set of linear equations between a finite set of expressions, each of which will be called a state. The states are labeled Q0, Q1, Q2, and so on. State Q0 is the expression 0, and Q1 is set to the input expression. Starting with Q0, each state is reduced using the method above.

When a state Q is reduced, it will be expressed as an equation of the form:

$$Q = \backslash Q \mid x1\ (x1\backslash Q) \mid x2\ (x2\backslash Q) \mid \ldots\ xn\ (xn\backslash Q)$$

Each of the expressions, x1\Q, x2\Q, ..., xn\Q is then considered to be a state and is added to the set of states if not already included.

In a better implementation, this algorithm can be run concurrently with an actual input file by working out the expansions x\Q on a "need to use" basis and leaving everything else alone. This is how the REX utility works. It's possible that an expression will blow up exponentially when represented by a DFA (such as the one in the input file "in2"), so it would be a waste to have to precalculate the whole automaton if the input the automaton is being used on is short.

5. Example

As an example, take this expression E = a* (b a*)*. When it is parsed, it is represented in the following tabular form:

	\E	E
	0	E0 = a
	1	E1 = E0*
	0	E2 = b
	0	E3 = E2 E1
	1	E4 = E3*
Q1	1	E = E5 = E1 E4
Q0	0	E6 = 0

(one observation alluded to here is that when looking at things as equations a parse tree is really nothing more than a finite set of non-recursive equations: a finite grammar). As shown above, each of the \E's is calculated on the fly, since these calculations are easy to carry out. Those expressions that represent states are marked by their labels (Q0, Q1, ...) to the left.

Reducing Q0 is a null operation so in reality this step is skipped. State Q1 is then reduced. The critical subterms of E5 are E4 (since \E1 = 1), and E1. The critical subterm of E4 is E3, and that of E3 is E2. E2 is reduced to the form:

$$E2 = b\ 1,$$

then:

$$E3 = E2\ E1 = b\ 1\ E1 = b\ E1,$$

then:

$$E4 = E3^* = 1 \mid b\ E1\ E3^* = 1 \mid b\ E1\ E4 = 1 \mid b\ E5$$

The critical subterm of E1 is E0, which reduces to the form:

$$E0 = a\ 1,$$

thus:

$$E1 = 1 \mid a\ 1\ E0^* = 1 \mid a\ E1$$

As per the table, the result of reducing E5 is:

$$E5 = 1 \mid b\ E5 \mid a\ E1\ E4 = 1 \mid a\ E5 \mid b\ E5$$

This results in the following updated table:

	\E	E
	0	E0 = a = a E7
	1	E1 = E0* = 1 | a E1
	0	E2 = b = b E7
	0	E3 = E2 E1 = b E1
	1	E4 = E3* = 1 | b E5
Q1	1	E = E5 = E1 E4 = 1 | a E5 | b E5
Q0	0	E6 = 0

1 E7 = 1

Since E5 is already a state, no new states are added and the calculation is complete. The finite automaton therefore has the form:

Q0 = 0

Q1 = 1 | a Q1 | b Q1

In this example, E (or Q1) is the only state in the DFA, so this is the minimal DFA.

6. Implementation

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

6.1. Parsing

The parser used was generated by Quantum Parsing which is based, in part, on the methods presented here. It was derived from the syntax (expressed as a recursive system of equations):

E → **0** | **1** | **x** | **(** E **)** | **[** E **]** | E **+** | E ***** | E **?** | E E | E **|** E | E **&** E | E **-** E | E **^** E,

start = (**x =** E **,**)* E,

with the natural precedence rules used for resolving ambiguities. This syntax results in a 2 state parser, with 1 ambiguous state resolved using an "action" table. The action table is a function of the form:

Lexical Item x Context --> Action

that takes a lexical item, matches it against the current context and determines from that what to do next.

Quantum Parsing is described in more detail in articles from early October of 1993 in comp.theory and comp.compilers ("The Equational Approach"), and mid February 1994 in comp.compilers ("Bottom up parsing by hand...", "Quantum Parsing...").

6.2. Expressing 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), i(E2))

i(E1 | E2) = f7(i(E1), i(E2))

i(E1 & E2) = f8(i(E1), i(E2))

i(E1 ^ E2) = f9(i(E1), i(E2))

i(E1 - E2) = f10(i(E1), i(E2))

where the ranges of the functions f2, f3, f4, f5, f6, f7, f8, f9, f10 and the values f0, f1 constitute a disjoint partition of the range of the index function i(E). The ranges of f0, f1, and f2 are not explicitly stored in the hash table. Instead, the expression for 0 and 1 are saved in a static location, and the expression corresponding to the symbol x is stored with the symbol.

For simplicity, f2, f3, f4, and f5 are chosen to be linear functions, and f6 and f7 are of the form:

f(DUP(a, b))

where f is linear, and where DUP maps: range(i) x range(i) onto range(i). In this implementation DUP(a, b) is taken to be the bitwise exclusive or of a and b.

This method of indexing has the effect of giving only expressions with a similar syntatic form the same index (which might come in handy in some future implementation).

This is all handled in the routine GetExp().

6.3. Sorting Terms

When expressions of the forms A B, A ^ B, A & B, and A | B are created, the terms in these expressions will be sorted, merged and/or grouped as per the outline given above in the rules (0)-(4). This is done in the case of A B by prefixing B by the terms in A, e.g.

a1 a2 ... an B = a1 (a2 (... (an B) ...))

in the routine CatExp(), and by merging the terms in A and B in the other cases in the routine BinExp(), e.g.:

(a ^ b ^ d) ^ (b ^ c ^ e) = a ^ b ^ b ^ c ^ d ^ e

(a | b | d) | (b | c | e) = a | b | c | d | e

The A - B operator is not touched.

6.4. Generating the DFA

The routine

FormState(Q)

will apply the reduction algorithm described above to the expression Q until normal form is reached, and then recursively apply the reduction to each of the resulting states x\Q, until all the states have been reduced.

6.5. Post-Processing

The DFA that is generated is usually close to being a mininal finite state automaton already. But in general not all equivalences are picked up during the reduction phase. Therefore a minimalization step will be carried out to group all the equivalent states. The state Q0 (which is the 0 expression) participates in this reduction. Any state found to be equivalent to Q0 corresponds to a non-terminating state in the corresponding finite automaton. In this way, all the non-terminating states can be found and removed (the difference operator A - B may create non-terminating states).

The routine MergeStates() carries out this process. The following recursive definition of equivalence is used by this algorithm:

A = B iff \A = \B and x\A = x\B for all x.

All states are initially treated as equivalent and considered so until otherwise proven. The algorithm itself does a pairwise comparison of states, and so is O(n^2). A better algorithm by Hopcroft can be implemented which would be O(n log n).

At the end, a stub routine WriteStates() will display the results in the format described above. This routine is what you would replace by an application, if you wanted to do something with the Regular Expression -> DFA conversion routine. The REX utility basically does just that.

7. Finiteness

A finiteness proof can be easily arrived at by carrying out an inductive argument in terms of the syntatic structure of the expression. For example, assuming that each of the expressions A and B have only a finite number of distinct derivatives, then since the derivatives of A B can be represented entirely in terms of these, they too will be finite in number.