AUTOMATIC CONSTRUCTION OF FINITE STATE MACHINES

Assume we start out with a regular grammar.

1. Determination of character classes:

Datastructure: A pair of sets L,S of character classes, this can be represented as a list of bitvectors with a bit set in each vector for each character in the class.

Algorithm 1.1.

1. Set the set L,S to null

2. Add each explicitly declared character class to the set L

3. Find each character c that occurs by itself as part of a regular expression and form a singleton class {c} which we add to L

4. $\forall_{i \in L} \forall_{j \in L}$*form* $k = i \cap j$, $m = i - k$, *n=j-k, and then set* $S \leftarrow S \cup$*{k,m,n}*

5. if $S \neq L$*Set* $L \leftarrow S, S \leftarrow \{\}$ *and goto 4*

We now have in L a set of character classes which can be used to pre process the input characters to obtain a reduced characterset. We associate with each characterclass in $c \in L$ an integer in the range $1..n$ where n is the numberof members of L. We now remap the grammar to be a grammar about a new alphabet - the integers $1..n$ replacing all occurences of single characters or character classes in the grammar with the appropriate class code. Assume the function $\kappa(x)$ maps a character class x to its code.

We initialise the character class table T as follows

Algorithm 1.2. $\forall_{i \in L} \forall_{c \in i} T_c \leftarrow \kappa(i)$

2. Construction of the finite state machine

We first present a naive algorithm and then look at problems that we would encounter using it.

Assume we have converted the grammar to one over the alphabet of character classes and that we have only the constructs

(), + , |, and sequencing in our grammar, Kleene star having been replaced by the rule $x\star \rightarrow (x+|)$. Assume further that all occurences of + are of the form (x)+ for some regular expression x.

Assume all alternations have been bracketed thus (a|b|c|d).

Where two alternatives have the same leading symbol as in: (ab|ac|de) rewrite this as (a(b|c)|de)

Data structures. A state transition table F [1..m,1..n] :1..m where m is the number of states supported

A string G representing the grammar

An integer a representing current state

Algorithm 2.1.

(1) *Asocciate each position in the string G with a numbered state in the table F*

(2) *set a to 1*
(3) *Move through the string incrementing a and as we do this do the following*
   (a) *If we have a pattern of the form* $(gs|bd|cf)$ *then fill in the state table to branch on the leading characters of the alternatives to the state corresponding to the next character of the alternative s, d, f in this case.*
   (b) *If we have a pattern of the form* $(g|bd)$ *then make the branch for the alternative starting with g go to the state corresponding to the closing bracket.*
   (c) *If we have a pattern of the form wxy and w is not the leading character of an alternative we fill in the table row for the position of w in the string G with a single valid branch to the state for the position of x in the string G and fill in all other alternatives with a branch to an error condition.*
   (d) *If we have a pattern of the form (pq)+ for some symbol p and regular expression q, then modify the state for the character position immediately following the + so that it branches on p to the state corresponding to the start of q.*

2.1. **Example.** take a grammar $(ab|ba)+ca$ where a,b,c are character classes

1	2	3	4	5	6	7	8	9
(	a	b	|	b	a	)	+	c

Generated state transition table

class	a	b	c
state			
1	3	6	0
3	0	7	0
6	7	0	0
7	3	6	99

Assume that 99 is the hit state.

2.2. **Problem.** This algorithm will work for most cases but has problems with examples like (xy)+xz which will recognise the sequences xyxyxz or xyxz.

Let us present the regular expression in a table showing positions

1	2	3	4	5	6	7
(	x	y	)	+	x	z

The problem here is that the state corresponding to the closing bracket(4) now has to branch two ways on x to position 3 and to position 7.

The way round this is to represent states not as single positions within the grammar but as a set of positions within the grammar and then assign finite state machine state numbers to these sets.

A set of positions in the grammar can be conveniently represented by a bitmap with a bit set for each grammar position that the machine could currently be at.

Data structures. A state transition table F [1..m,1..n] :1..m where m is the number of states supported

A string G representing the grammar

An array of bits M[1..m,1..g]

An integer s representing the current state, and a variable t to represent the next free state.

An integer b for iterating through bitmaps

Algorithm 2.2.

(1) *s to 1, tto 2 and clear M*
(2) *set M[1,1]=1*
(3) *repeat until M[s] does not change : For b = 1 to g if M[s]*
   (a) *If the grammar string starting at G[b] is of the form*

b	*b+1*	*b+2*	*b+3*	*b+4*	*b+5*	*b+6*
(	*X*	*/*	*Y*	*/*	*Z*	*)*

   then set bits b+1, b+3,b+5 in M[s]. Make the obvious adjustments to the offsets here where X,Y,Z are more than 1 character long.
   (b) *if the grammar string around at G[b] is of the form*

b-2	*b-1*	*b*	*b+1*	*b+2*
(	α	*)*	*+*	β

   then set bits b+2 and b-2 of M[s]: $M[s, b+2] \leftarrow 1, M[s, b-2] \leftarrow 1$
   (c) *if the grammar string around at G[b] is of the form*

b	*b+1*	*b+2*
(	α	*)*

   then set the appropriate bit of the bitmap of M[s,b+1] $\leftarrow 1$.
(4) *For b = 1 to g if M[s,b]*
   If the grammar string starting at G[b] is of the form $\alpha\beta$ *for some terminal symbol* α *and some regular expression* β*, then*
   (a) *put a branch into F so that F[s,* α*]*$\leftarrow t$
   (b) *set one bit in the row of M corresponding to the start of* β*,* $M[t, b+1] \leftarrow 1$ *to seed the processing of the destination*
   (c) $t \leftarrow t+1$ *to ensure that this table row is not re-used.*
(5) *increment s*
(6) *if s=t terminate*
(7) *go back to step 3*

The extensions to the algorithm necessary to get the finite state machine to return lexeme codes are pretty simple. One combines the regular expessions for all the lexemes using | into a lager regular expression and ensures that the terminating states of all these top level expressions are marked as hits in an action table that is constructed in parallel. to the table F. The combination of the state and the branch taken on a hit can be used to index a table specifying the output code to be produced by the FSM.