LR(0) Grammars

The starting point for LR(0) grammars is weak precedence grammars, extended to include the rule

S' ::= S EOF

where S is the start rule. We will also include a new stack symbol, $, denoting the bottom of the stack. We will do several examples with the following simple grammar:

(P1): S' ::= S EOF	$\mathcal{L}$(S')={S,a,d}	$\mathcal{R}$(S')={EOF}
(P2): S ::= aSb	$\mathcal{L}$(S)={a,d}	$\mathcal{R}$(S)={b,c}
(P3): S ::= aSc		
(P4): S ::= db		

This grammar has the following precedence table:

	a	b	c	d	S	EOF
a	<			<	=	
b		>	>			>
c		>	>			>
d		=				
S		=	=			=
$	<			<	<	

Note that this makes Table[$, y] = "<" for any y in ℒ(S'), for these are the only symbols that could be pushed onto an empty stack.

LR(0) grammar write this table as

	a	**b**	**c**	**d**	**S**	**EOF**
a	sh			sh	sh	
b		red	red			red
c		red	red			red
d		sh				
S		sh	sh			acc
$	sh			sh	sh	

where the table entries are
sh for "shift"
red for "reduce"
and acc for "accept"

Remember that our grammar is

(P1): S' ::= S EOF $\mathcal{L}$(S')={S,a,d} $\mathcal{R}$(S')={EOF}

(P2): S ::= aSb $\mathcal{L}$(S)={a,d} $\mathcal{R}$(S)={b,c}

(P3): S ::= aSc

(P4): S ::= db

Note that if we reduce with b on top of the stack it could only come from (P2) or (P4). When we shift that b onto the stack, if it goes on top of an S we could shift b_2; if on top of a d we could shift b_4.

(P1): S' ::= S EOF $\mathcal{L}$(S')={S,a,d} $\mathcal{R}$(S')={EOF}

(P2): S ::= aSb $\mathcal{L}$(S)={a,d} $\mathcal{R}$(S)={b,c}

(P3): S ::= aSc

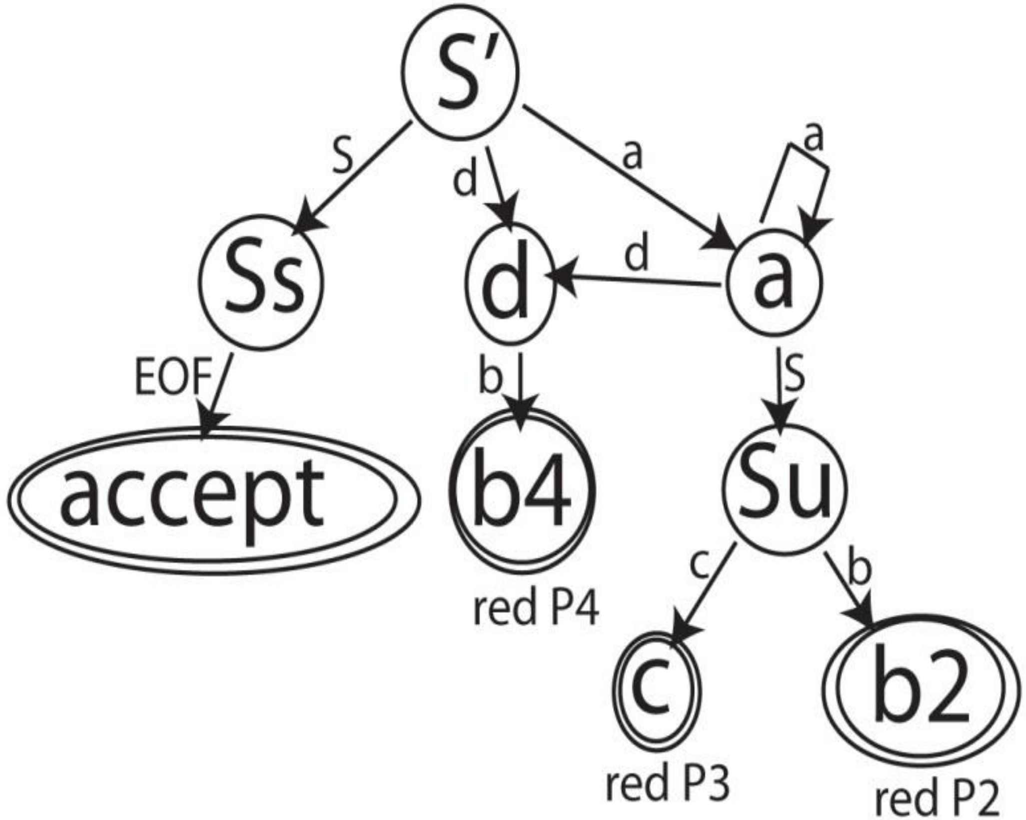
(P4): S ::= db

Note also that there could be two kinds of S values to push on the stack. If we push an S onto an empty stack, we only need the EOF token to accept the input string. We will call S in this case a "satisfied S", or S_s. If we push an S onto a non-empty stack we will call it an "unsatisfied S", or S_u. In this way our stack tokens can tell us something about what is on the stack below them.

Our action table is now:

	a	**b**	**c**	**d**	**S**	**EOF**
a	sh a			sh d	sh S_u	
b_2		red P2	red P2			red P2
b_4		red P4	red P4			red P4
c		red P3	red P3			red P3
d		sh b_4				
S_u		sh b_2	sh c			
S_s						acc
$	sh a			sh d	sh S_s	

Note that we could encode the action table as a DFA. The terminal states do reductions.



This automaton has all of the information of the table with one exception -- it does reductions without checking that the next token is appropriate. This is okay, because the error will be detected before the next token is shifted onto the stack.

Building the LR(0) Action Table

We will usually reverse the steps of the last example. Instead of producing a DFA from the action table, we will have an algorithm that produces a DFA from the grammar, and we will use this DFA to derive the action table that we actually use in parsing.

Def. An LR(0) item is a grammar rule with a dot on the right-hand side, as in [A ::= X.Y]. The item [A ::= X.Y] means that we have seen X and are expecting Y to allow a reduction to A.

Each state of our DFA wil consist of a collection of LR(0) items.

To find the states of the DFA, begin with the item [S' ::= .S EOF], where S is the start state. For any item in a state with the dot preceding a non-terminal symbol, as in [A ::= α.Bβ], we add in all of the items with that non-terminal on the left, as in [B ::= .χ]
If a state has the item [A ::= α.xβ] we draw an edge labeled x to the state containing [A ::= αx.β] If this leaves the dot before a non-terminal we expand the new state to include all of the items derived from that non-terminal.

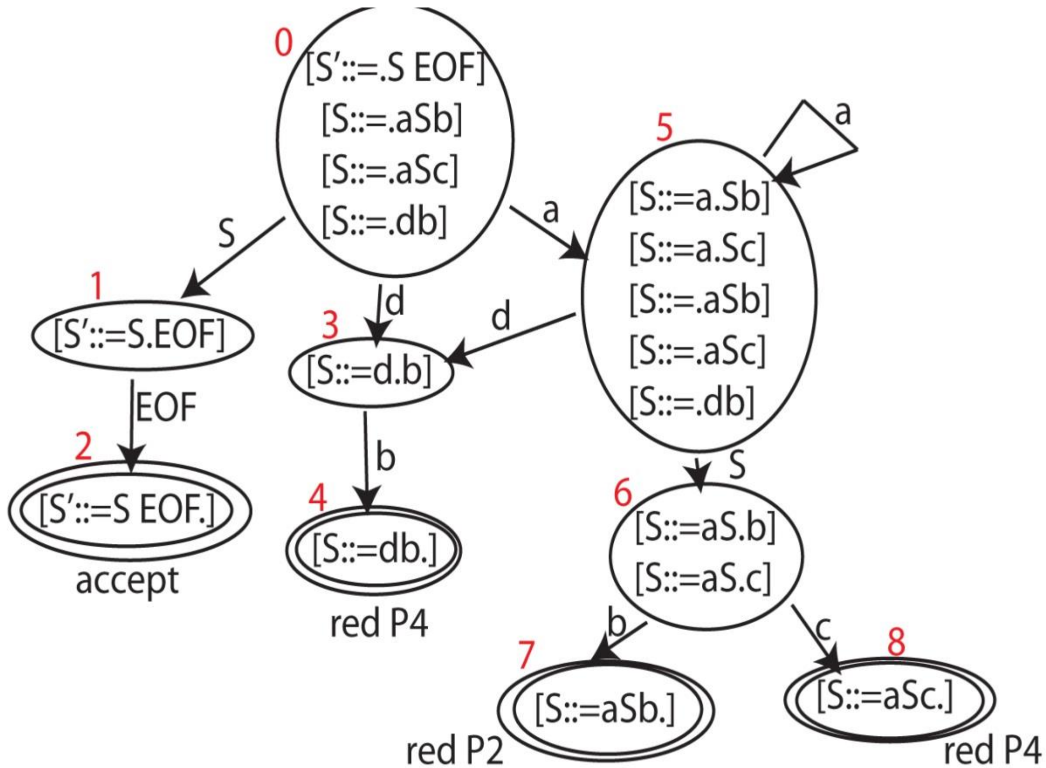
Here is the DFA we get for our grammar

(P1): S' ::= S EOF

(P2): S ::= aSb

(P3): S ::= aSc

(P4): S ::= db



Note that in our DFA we have numbered the states so we have a way to refer to them.

We will modify our action table so the rows are indexed by states; the columns will still be indexed by terminal and non-terminal symbols.

An edge in the DFA from state i: [A::=α.xβ] to state j: [A::=αx.β] is represented in the table by Table[i,x] = sh j (shift x, enter state j).

A reduction state for rule Pk: [A::=a.] is represented in the table by Table[i,x]= red k for every x.

This gives the following action table:

	a	b	c	d	S'	S	EOF
0	sh 5			sh 3		sh 1	
1							sh 2
2	acc	acc	acc	acc	acc	acc	acc
3		sh 4					
4	red 4	red 4	red 4	red 4	red 4	red 4	red 4
5	sh 5			sh 3		sh 6	
6		sh 7	sh 8				
7	red 2	red 2	red 2	red 2	red 2	red 2	red 2
8	red 3	red 3	red 3	red 3	red 3	red 3	red 3

To use such a table, start in state 0. On each shift push the new state (from the table) on top of the shifted symbol. On a reduction pop the symbols off the stack and use the uncovered state along with the symbol being pushed to determine the new state to enter.