Constructing an SLR parse table

This document was created by Sam J. Frazier based on class lectures by Professor Carol Zander.
The document was edited by Carol Zander.

S simple
L left-to-right scan of input
R rightmost derivation in reverse

Part 1: Create the set of LR(0) states for the parse table

For the rules in an augmented grammar, G', begin at rule zero and follow the steps below:

State creation steps (big picture algorithm)

1. Apply the ***start operation*** and
2. ***complete the state:***
   a. Use one ***read operation*** on each item C (non-terminal or terminal) in the current state to create more states.
   b. Apply the ***complete operation*** on the new states.
   c. Repeat steps a and b until no more new states can be formed.

Operations defined

A, S, X: non-terminals
w,x,y,z: string of terminals and/or non-terminals
C: one terminal or one non-terminal

start: if S is a symbol with [S -> w] as a production rule, then [S -> .w} is the item associated with the start state.

read: if [A --> x.Cz] is an item in some state, then [A --> xC.z] is associated with some other state. When performing a read, all the items with the dot before the same C are associated with the same state. (Note that the dot is before anything, either terminal or non-terminal.)

complete: if [A --> x.Xy] is an item, then every rule of the grammar with the form [X --> .z] must be included within this state. Repeat adding items until no new items can be added. (Note that the dot is before a non-terminal.)

Example for Part 1

Consider the augmented grammar G':

0. S' --> S$
1. S --> aSbS
2. S --> a

The set of LR(0) item sets, the states:

State	Item	Notes
I_0	[S' -> .S$]	start operation; read on S goes to I_1 (state 1)
	[S -> .aSbS]	complete operation on S rule; read on 'a' goes to I_2 (state 2)
	[S -> .a]	continue complete for all rules 'S'; ditto the read on 'a', to state 2
I_1	[S' -> S.$]	read on 'S' from first line; Note: never read on '$' nothing to read on; nothing to complete
I_2	[S -> a.SbS]	from read on 'a' from state I_0; read on 'S' goes to I_3 (state 3)
	[S -> a.]	continue from read on 'a' from state I_0 (see step 2 of state creation) nothing to read on; nothing to complete
	[S -> .aSbS]	complete the state because of '.S' in the first item read on 'a' cycles back to state 2
	[S -> .a]	continue complete of all grammar rules for 'S' ditto read on 'a' cycles back to state 2
I_3	[S -> aS.bS]	from read on 'S' from state I_2 the dot is before a non-terminal, no complete operation read on 'b' goes to I_4 (state 4)
I_4	[S -> aSb.S]	from read on 'b' from state I_3; read on 'S' goes to I_5 (state 5)
	[S -> .aSbS]	complete the state because of '.S' in the first item; note: dot always in front for completes read on 'a' cycles back to state 2
	[S -> .a]	continue complete; ditto read on 'a' cycles back to state 2
I_5	[S -> aSbS.]	from read on 'S' from state 5; nothing to read on

Part 2: Construct the parse table

To create the parse table, you need the FIRST and FOLLOW sets for each non-terminal in your grammar. Also, you need the completed set of state items from part 1.

Now, draw an *n x m* table for your parse table, and label it appropriately. The *n* is equal to the number of states you have from part 1. You determine the *m* by counting all non-terminals and all terminals in the grammar. Provide a row or column for each *n* and *m* item.

Label each row *n* with a state number, starting at zero. Label each column *m* with one terminal per column, starting from left to right. After labeling with all the terminals, label the remaining columns with the non-terminals.

The group of columns on the left (terminals) is the ACTION side. The group of columns on the right (non-terminals) is the GOTO side.

To fill in the table, you follow these four rules.

<u>Construction rules</u>

α, β = any string of terminals and/or non-terminals
X, S', S = non-terminals

(When dot is in middle)

1. if [A --> α.aβ] ε I_i and read on 'a' produces I_j then ACTION [i , a] = SHIFT j.
2. if [A --> α.Xβ] ε I_i and read on 'X' produces I_j then GOTO [i , X] = j.

(When dot is at end)

3. if [A --> α.] ε I_i then ACTION [i , a] = REDUCE on A -> α for all a ε FOLLOW(A).
4. if [S' --> S.] ε I_i then ACTION [i , $] = ACCEPT.

For the example, the parse table:

	a	**b**	**$**		**S**
0	s2				1
1			accept		
2	s2	r2	r2		3
3		s4			
4	s2				5
5		r1	r1		

Using the parse table construction rules for the augmented grammar G':

0. S' --> S$
1. S --> aSbS
2. S --> a

The FIRST and FOLLOW statements are:

FIRST(S) = {a}

FOLLOW(S) = {b, $}

State	Item	Notes
		(Remember that a SHIFT refers to state, REDUCE refers to grammar rule)
I_0	`[S' -> .S$]`	read on S goes to state 1; dot in middle #2, GOTO[0,S] = 1
	`[S -> .aSbS]` `[S -> .a]`	read on 'a' for both of these goes to state 2; dot in middle #1, ACTION[0,a] = SHIFT 2
I_1	`[S' -> S.$]`	dot at end #4 (only one of these), ACTION[1,$] = REDUCE 2
I_2	`[S -> a.SbS]`	read on 'S' goes to state 3; dot in middle #2, GOTO[2,S] = 3
	`[S -> a.]`	dot at end #3, ACTION[2,b] = ACTION[2,$] = REDUCE 2
	`[S -> .aSbS]` `[S -> .a]`	read on 'a' for both of these cycles back to state 2; dot in middle #1, ACTION[2,a] = SHIFT 2
I_3	`[S -> aS.bS]`	read on 'b' goes to state 4; dot in middle #1, ACTION[3,b] = SHIFT 4
I_4	`[S -> aSb.S]`	read on 'S' goes to state 5; dot in middle #2, GOTO[4,S] = 5
	`[S -> .aSbS]` `[S -> .a]`	read on 'a' cycles for both of these cycles back to state 2; dot in middle #1, ACTION[4,a] = SHIFT 2
I_5	`[S -> aSbS.]`	dot at end #3, ACTION[5,b] = ACTION[5,$] = REDUCE 1

	a	**b**	**$**		**S**
0	s2				1
1			accept		
2	s2	r2	r2		3
3		s4			
4	s2				5
5		r1	r1		