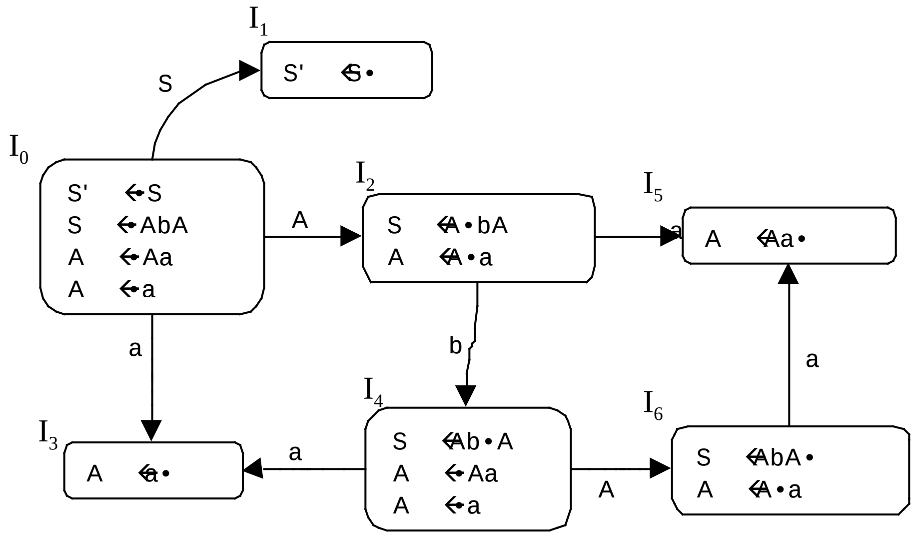
PS2 Solutions

1) a) Here is the family of SLR(1) configurating sets.



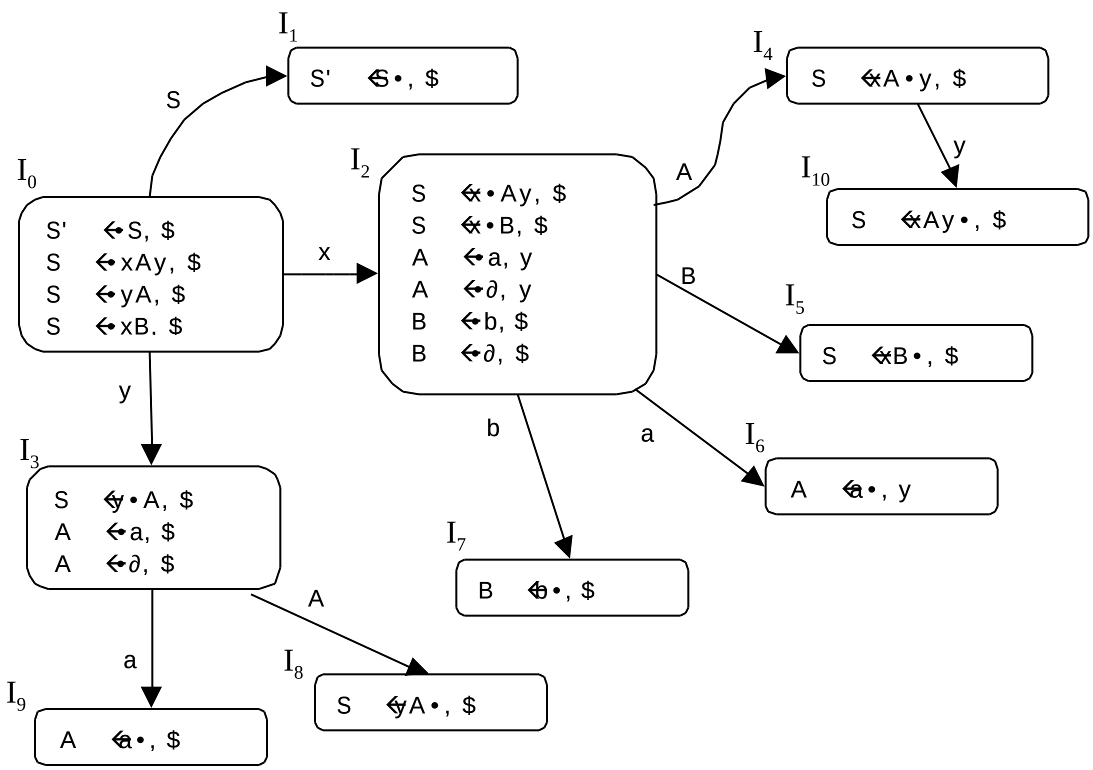
b) Here is the trace of the SLR(1) parser on the input aaba.

STACK STACK	REMAINING INPUT	PARSER ACTION
S_0	aaba$	Shift S_3
S_0S_3	aba$	Reduce A → a, goto S_2
S_0S_2	aba$	Shift S_5
$S_0S_2S_5$	ba$	Reduce A → Aa, goto S_2
S_0S_2	ba$	Shift S_4
$S_0S_2S_4$	a$	Shift S_3
$S_0S_2S_4S_3$	$	Reduce A → a, goto S_6
$S_0S_2S_4S_6$	$	Reduce S → AbA, goto S_1
S_0S_1	$	Accept

2) Here is the production numbering we used:

0) S' ← S	2) S ← yA	4) A ← a	6) B ← b
1) S ← xAy	3) S ← xB	5) A ← ∂	7) B ← ∂

Here are our configurating sets in goto-graph form (you didn't have to show this step, but we figure it will help you to interpret our table below):



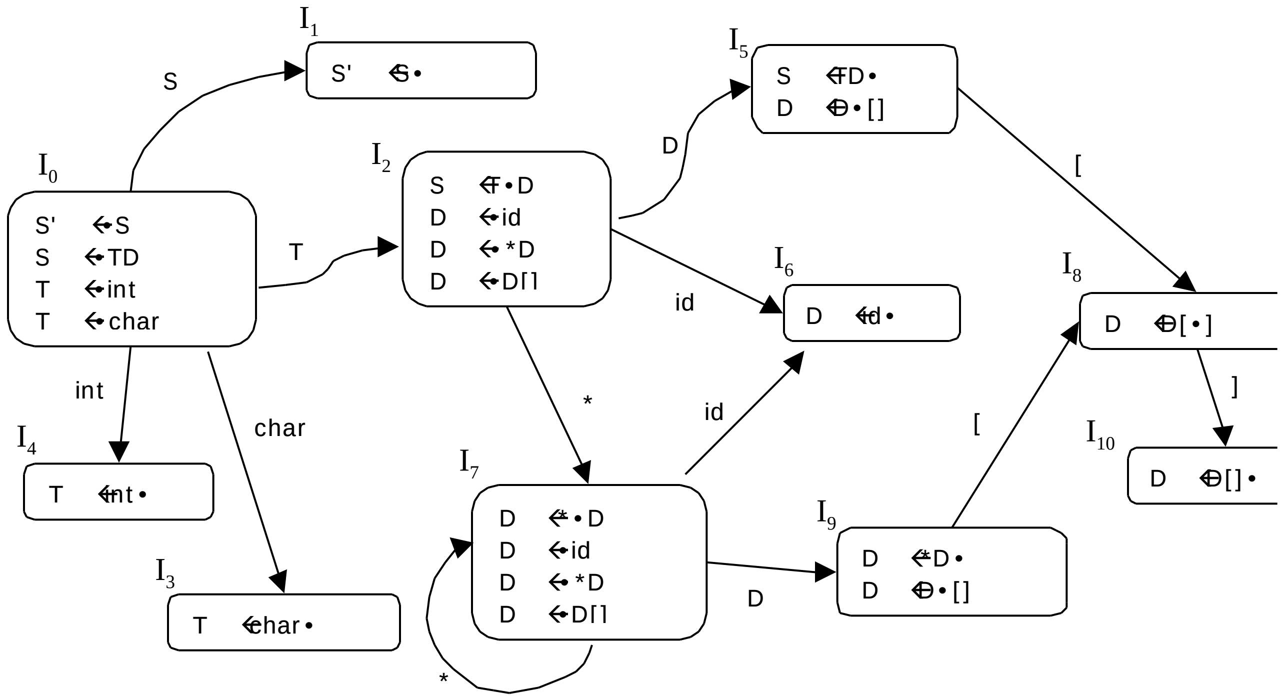
a) Here is the LR(1) parsing table:

State on top of stack	Action					Goto		
	a	b	x	y	$	S	A	B
0			s2	s3		1		
1					accept			
2	s6	s7		r5	r7		4	5
3	s9				r5		8	
4				s10				
5					r3			
6				r4				
7					r6			
8					r2			
9					r4			
10					r1			

b) No. In state 2 there are two completed items that can be reduced: A ← ∂ and B ← ∂. SLR uses only the follow set to determine if the reduction is valid. Follow(A) = { y $ } and Follow(B) = { $ }. The SLR(1) parser would have a conflict if the next input was $.

c) Yes. The only candidate states for merging are states 6 and 9. The merged lookaheads do not introduce any new reduce-reduce conflicts.

3) The SLR configurating sets, provided as an illustration to help you understand our answer. (I just so love creating these diagrams I couldn't resist making another…?)



- **a)** There is one conflict; a shift-reduce conflict in state 9. `Follow(D)` is `{ $ [ }` so we could either reduce or shift if the next input is an open bracket. Should we reduce the `*D` on top of the stack or should we shift the `[` and keep going? Parsing the input `int *arr[]` would exhibit the conflict. Note there is no conflict in state 5, `Follow(S)` is just `{ $ }` and thus input `[` always shifts in that state.
- **b)** Adding lookaheads does not resolve the issue because `[` is a lookahead in the LR(1) configurating set for state 9 as well. This grammar is neither LALR(1) nor LR(1). The fundamental problem is that the grammar is ambiguous. Does the declaration `int *arr[]` declare `arr` as an array of int pointers or a pointer to an int array? It might be helpful to draw the two different parse trees that would result. The C language takes the first interpretation, which means that brackets bind to the identifier first, then the star is added. You could re-write the grammar and add a new non-terminal to enforce this precedence or resolve the conflict by always shifting on `[` in state 9. If you examine the productions for D, there is both a right-recursive and a left-recursive expansion. Do you see why a non-terminal with such productions will always result in an ambiguous grammar?

4) In a shift/reduce conflict, there is a sequence on top of the stack that is a valid handle to reduce that is also a substring of another longer handle that could be shifted. Like the "maximal munch" strategy taken by lex, yacc prefers matching a longer pattern over a shorter, and will shift. Choosing reduction instead is likely to change the language accepted because it effectively disallows the longer sequence to be built. Visualize the goto-graph: any successors that follow the shift action have been excised unless there is an alternate path to get to those states! Without any other information to go on, yacc assumes this isn't what you wanted.

The dangling-else ambiguity is just one example: `S → iSeS | iS`. The shift/reduce conflict occurs after parsing an `iS` followed by an `e`. Yacc chooses to shift, binding the else to the inner if. If instead we always reduce `iS`, we would never be able to recognize the longer string `iSeS` because the first `iS` would be reduced to `S` and we would now have an `S` on the stack with `eS` coming up in the input that matches no possible right side production. We would get an error and the parser no longer recognizes the correct language.

Forcing a shift can also change the language (consider `S` → `a | abc | Sb` on input `ab`) but it is much less likely (I had to work at building an example that fails when you force shift, whereas most grammars with a shift-reduce conflict are amputated by a forced reduction).

For some grammars (the ambiguous expression grammar, the ambiguous declarator grammar in problem 3), forcing a shift or forcing a reduce doesn't change the language accepted (there is any alternate path to get the part of the goto-graph in either way), but forcing one path removes the alternate route, thus removing the ambiguity.

5) Consider two equivalent tiny grammars, one left-recursive: `S` → `Sx | x` and one right: `S` → `xS | x`. There are the same number of states in the LR(1) parsers for the two grammars so there is no advantage in the runtime memory requirements for either table. Consider parsing the input `xxx…x` with a thousand `x`'s. In the left-recursive form, the parser shifts the first `x`, immediately reduces it to `S`, shifts another `x`, and then reduces `Sx` to `S`. It does this again and again for each subsequent `x`. The parse stack grows and shrinks, only getting three deep at its maximum. For the right-recursive grammar, the parser shifts the first `x`, then shifts the second `x`, and so on. The parser doesn't reduce until it gets to the `$` at the end of the input. It then reduces the last `x` on top of the stack to `S`, then reduces the `xS` on the stack to `S`, and again and again until the stack is empty. The stack had a thousand states on it by the time we got to the end of the input! Although the LR parsing can handle either, clearly the left-recursive grammar is handled more efficiently.

There were a few of you who seemed to have some misunderstanding about how the stack works. Symbols that are popped off of the stack are not pushed back on later, they are replaced by the non-terminal on the left side of the production being reduced. Also we only consider the symbols on the top of the stack when trying to form a handle to reduce, the parser does not search through the stack for matches.

To the optimist, the glass is half full. To the pessimist, the glass is half empty. To the engineer, the glass is twice as big as it needs to be.