Compiler Design

Spring 2011

Syntactic Analysis

Sample Exercises and Solutions

Prof. Pedro C. Diniz

USC / Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, California 90292
pedro@isi.edu

Problem 1: Give the definition of Context Free Grammar (CFG).

Solution: The tuple G = {NT, T, S ∈ NT, P: NT → (NT ∪ T)*}, i.e., a set of non-terminal variable symbols, a set of terminal symbols (or tokens), a start symbol from the set of non-terminals and a set of productions that map a given non-terminal to a finite sequence of non-terminal and terminal symbols (possibly empty).

Problem 2: Argue that the language of all strings of the form {{...{}...}} (equal number of '{' and '}') is not regular. Give CFG that accepts precisely that language.

Solution: If it were regular there would be a DFA that would recognize it. Let's suppose that there is such a machine M. Given that the length of the input string can be infinite and that the states of M are in finite number, then, there must be a subsequence of the input string that leads to a cycle of states in M. Without loss of generality we can assume that that substring that induces a cycle in the states of M has only '{' (we can make this string as long as you want). If in fact there were such a machine that could accept this long string then it could also accept the same string plus one occurrence of the sub-string (idea of the pumping lemma). But since this sub-string does not have equal number of '{' and '}' then the accepted string would not be in the language contradicting the initial hypothesis. No such M machine can therefore exist. In fact this language can only be parsed by a CFG. Such CFG is for example, S → { S } | e where e is the epsilon or empty string.

Problem 3: Consider the following CFG grammar,

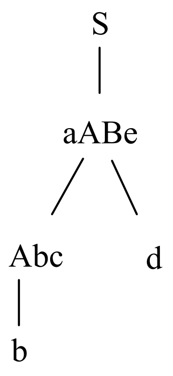
S → aABe
A → Abc | b
B → d

where 'a', 'b', 'c' and 'd' are terminals, and 'S' (start symbol), 'A' and 'B' are non-terminals.

a) Parse the sentence "abbcde" using right-most derivations.
b) Parse the sentence "abbcde" using left-most derivations.
c) Draw the parse tree.

Solution:

a) S → aABe → aAde → aAbcde → abbcde
b) S → aABe → aAbcBe → abbcBe → abbcde
c) Shown below:



Problem 4: Consider the following (subset of a) CFG grammar

stmt	$\rightarrow$ NIL | stmt ';' stmt | ifstmt | whilestmt
ifstmt	$\rightarrow$ IF bexpr THEN stmt END
whilestmt	$\rightarrow$ WHILE bexpr DO stmt END

where NIL, ';', IF, THEN, END, WHILE and DO are terminals, and "stmt", "ifstmt", "whilestmt" and "bexpr" are non-terminals.

For this grammar answer the following questions:

a) Is it ambiguous? Is that a problem?
b) Design a non-ambiguous equivalent (subset of a) grammar.

Solution:

a) Is it ambiguous? Why? Is that a problem?

Yes, this language is ambiguous as there are two distinct parse trees for a specific input string. For example the input

b) Design a non-ambiguous equivalent (subset of a) grammar.

Problem 5: Consider the following Context-Free Grammar G = ({S,A,B},S,{a,b},P) where P is

$S \rightarrow AaAb$
$S \rightarrow Bb$
$A \rightarrow \varepsilon$
$B \rightarrow \varepsilon$

(a) Compute the FIRST sets for A, B, and S.
(b) Compute the FOLLOW sets for A, B and S.
(c) Is the CFG G LL(1)? Justify

Solution:

a. The FIRST of a sentential form is the set of terminal symbols that lead any sentential from derived from the very first sentential form. In this particular case A and B only derive the empty string and as a result the empty string is the FIRST set of both non-terminal symbols A and B. The FIRST of S, however, includes "a" as in the first production once can derive a sentential forms that starts with an "a" given that A can be replaced by the empty string. A similar reasoning allows you to include "b" in the FIRST(S). In summary: FIRST(A) = {ε}, FIRST(B) = {ε}, FIRST(S) = {a,b}

b. The FOLLOW set of a non-terminal symbo is the set of terminals that can appear after that non-terminal symbol in any sentential form derived from the grammar's start symbol. By definition the follow of the start symbol will automatically include the $ sign which represents the end of the input string. In this particular case by looking at the productions of S one can determine right away that the follow of A includes the terminal "a" abd "b" and that the FOLLOW of B includes the terminal "b". Given that the non-terminal S does not appear in any productions, not even in its own productions, the FOLLOW of S is only \$. In summary: FOLLOW(S) = {\$}, FOLLOW(A) = {a,b}, FOLLOW(B) = {b}.

c. YES, because the intersection of the FIRST for every non-terminal symbol in empty. This leads to the parsing table for this LL method as indicated below. As there are no conflict in this entry then grammar is clearly LL(1).

	a	b	\$
S	$S \to AaAb$	$S \to Bb$	
A	$A \to \varepsilon$	$A \to \varepsilon$	
B		$B \to \varepsilon$	

Problem 6: Construct a table-based LL(1) predictive parser for the following grammar G = {bexpr, {bexpr, bterm, bfactor},{not, or, and, (,), true, false}, P } with P given below.

bexpr $\to$ bexpr or bterm | bterm
bterm $\to$ bterm and bfactor | bfactor
bfactor $\to$ not bfactor | (bexpr) | true | false

For this grammar answer the following questions:

(a) Remove left recursion from G.
(b) Left factor the resulting grammar in (a).
(c) Computer the FIRST and FOLLOW sets for the non-terminals.
(d) Construct the LL parsing table.
(e) Verify your construction by showing the parse tree for the input string "true or not (true and false)"

Solution:

(a) Removing left recursion:

bexpr $\to$ bterm E'

E' $\to$ or bterm E'
$\to$ ε

T' $\to$ and bfactor T'
$\to$ ε

bterm $\to$ bfactor T'

bfactor $\to$ not bfactor
| (bexpr)
| true
| false

(b) Left factoring: The grammar is already left factored.

(c) First and Follow for the non-terminals:

First(bexpr)	= First(bterm) = First (bfactor) = {not, (, true, false}
First(E’)	= {or, ε}
First(T’)	= {and, ε}
Follow(bexpr)	= {$,)}
Follow(E’)	= Follow(bexpr) = {$,)}
Follow(bterm)	= First(E’) ∪ Follow(E’) = {or,), $}
Follow(T’)	= Follow(bterm) = {or,), $}
Follow(bfactor)	= First(T’) ∪ Follow(T’) = {and, or,), $}

(d) Construct the parse table:

	or	and	not	(	)	True/false	$
bexpr			bexpr → bterm E’	bexpr → bterm E’		bexpr → bterm E’	
E’	E’ → or bterm E’				E’→ ε		E’ → ε
bterm			bterm → bfactor T’	bterm → bfactor T’		bterm → bfactor T’	
T’	T’ → ε	T’ → and bfactor T’			T’→ ε		T’→ ε
bfactor			bfactor → not bfactor	bfactor → (bexpr)		bfactor → true/false	

(e) To verify the construction, at each point we should find the right production and insert it into the stack. These productions define the parse tree. Starting from the initial state and using the information in the parse table:

Stack	Input	Production
$	true or not (true and false)	bexpr-> bterm E’
$E’ bterm	true or not (true and false)	bterm-> bfactor T’
$E’ T’ bfactor	true or not (true and false)	bfactor-> true
$E’ T’ true	true or not (true and false)	
$E’ T’	or not (true and false)	T’->epsilon
$E’	or not (true and false)	E’->or bterm E’
$E’ bterm or	or not (true and false)	
$E’ bterm	not (true and false)	bterm-> bfactor T’
$E’ T’ bfactor	not (true and false)	bfactor-> not bfactor
$E’ T’ bfactor not	not (true and false)	
$E’ T’ bfactor	(true and false)	bfactor-> (bexpr)
$E’ T’) bexpr (	(true and false)	
$E’ T’) bexpr	true and false)	bexpr-> bterm E’
$E’ T’) E’ bterm	true and false)	bterm-> bfactor T’
$E’ T’) E’ T’ bfactor	true and false)	bfactor-> true
$E’ T’) E’ T’ true	true and false)	
$E’ T’) E’ T’	and false)	T’->and bfactor T’
$E’ T’) E’ T’ bfactor and	and false)	
$E’ T’) E’ T’ bfactor	false)	bfactor-> false
$E’ T’) E’ T’ false	false)	
$E’ T’) E’ T’	)	T’->epsilon
$E’ T’) E’	)	E’->epsilon
$E’ T’)	)	
$E’ T’	$	T’->epsilon
$E’	$	E’->epsilon
$	$	

bexpr-> bterm E’ -> bfactor T’ E’ -> true T’ E’ -> true E’ -> ture or bterm E’ -> true or bfactor T’ E’ -> true or not bfactor T’ E’ -> true or not (bexpr) T’ E’-> true or not (bterm E’) T’ E’ -> true or not (bfactor T’ E’) T’ E’ -> true or not (true T’ E’) T’ E’ -> true or not (true and bfactor T’ E’) T’ E’ -> or not (true and false T’ E’) T’ E’ -> or not (true and false E’) T’ E’-> or not (true and false) T’ E’-> or not (true and false) E’ -> or not (true and false)

So there is a leftmost derivation for the input string.

Problem 7: Given the CFG G = {S, {S, U,V, W}, {a, b,c,d}, P } with P given as shown below:

S → UVW
U → (S) | aSb | d
V → aV | ε
V → cW | ε

a) Construct its a table-based LL(1) predictive parser;
b) Give the parsing actions for the input string "(dc)ac".

Solution:

a) This grammar is not left-recursive and not left-factored so there is no need to modify it. We begin by computing the FIRST and FOLLOW sets of each of the non-terminals.

FIRST(S) = { (, a, d }	FOLLOW(S) = {), b, $ }
FIRST(U) = { (, a, d }	FOLLOW(U) = { a, c,), b, $ }
FIRST(V) = { a, ε }	FOLLOW(V) = { c,), b, $ }
FIRST(W) = { c, ε }	FOLLOW(W) = {), b, $ }

Based on these sets we fill-in the entries M[T,t] of the LL parse table using the three rules:

1. If t ∈ T and t ∈ FIRST(α), for S → α ∈ P then M[S,t] = S → α
2. If A → ε then M[A,a] = A → ε for all a ∈ FOLLOW(A)
3. Otherwise M[T,t] is error.

Given these rules we obtain the LL(1) parsing table below which has no conflicts.

M[T,t]	**a**	**b**	**c**	**d**	**(**	**)**	**$**
S	S → UVW			S → UVW	S → UVW		
U	U → aSb			U → d	U → (S)		
V	V → aV	V → ε	V → ε			V → ε	V → ε
W		W → ε	W → cW			W → ε	W → ε

b) For the string "(dc)ac" we begin with the stack with the start symbol, in this case S, and expand it using the table above.

Stack	**Input**	**Comment**
$S	(dc)ac$	Expanded using production S → UVW as (∈ FIRST(UVW)
$WVU	(dc)ac$	Expanded using production U → (S) as (∈ FIRST(U)
$WV)S(	(dc)ac$	Match (, advance the input pointer and pop (from the stack
$WV)S	dc)ac$	Expanded using production S → UVW as d ∈ FIRST(UVW)
$WV)WVU	dc)ac$	Expanded using production U → d as d ∈ FIRST(U)
$ WV)WVd	dc)ac$	Match d, advance the input pointer and pop d from the stack
$ WV)WV	c)ac$	Using production V → ε as c ∈ FOLLOW(V)
$ WV)W	c)ac$	Expanded using production W → cW as c ∈ FIRST(cW)
$ WV)Wc	c)ac$	Match c, advance the input pointer and pop c from the stack
$ WV)W	)ac$	Using production W → ε as) ∈ FOLLOW(W)
$ WV)	)ac$	Match), advance the input pointer and pop) from the stack
$ WV	ac$	Expanded using production V → aV as a ∈ FIRST(aV)
$ WVa	ac$	Match a, advance the input pointer and pop a from the stack
$ WV	c$	Using production V → ε as c ∈ FOLLOW(V)
$ W	c$	Expanded using production W → cW as c ∈ FIRST(cW)
$ Wc	c$	Match c, advance the input pointer and pop c from the stack
$W	$	Using production W → ε as $ ∈ FOLLOW(W)
$	$	**Accept!**

Problem 8: Consider the following grammar for variable and class declarations in Java:

```
<Decl>      → <VarDecl>
            | <ClassDecl>
<VarDecl>   → <Modifiers> <Type> <VarDec> SEM
<ClassDecl> → <Modifiers> CLASS ID LBRACE <DeclList> RBRACE
<DeclList>  → <Decl>
            | <DeclList> <Decl>
<VarDec>    → ID
            | ID ASSIGN <Exp>
            | <VarDec> COMMA ID
            | <VarDec> COMMA ID ASSIGN <Exp>
```

For this grammar answer the following questions:

a. Indicate any problems in this grammar that prevent it from being parsed by a recursive-descent parser with one token look-ahead. You can simply indicate the offending parts of the grammar above.

b. Transform the rules for <VarDec> and <DeclList> so they can be parsed by a recursive-descent parser with one token look-ahead i.e., remove any left-recursion and left-factor the grammar. Make as few changes to the grammar as possible. The non-terminals <VarDec> and <DeclList> of the modified grammar should describe the same language as the original non-terminals.

Solution:

b. This grammar is left recursive which is a fundamental problem with recursive descendent parsing either implemented as a set of mutually recursive functions of using a tabel-driven algorithm implementation. The core of the issue hás to deal with the fact that when this parsing algorithm tries to expand a production with another production that starts (either by direct derivation or indirect derivation) with the same non-teminal that was the leading (or left-most non-terminal) in the original sentential form, it will have not onsumed any inputs. This means that it can reapply the same derivation sequence without consuming any inputs and continue to expand the sentential form. Given that the size of the sentential fom will have grown and no input tokens will have been consumed the process never ends and the parsing eventually fails due to lack of resources.

c. We can apply the immediate left-recursion elimination technique for <VarDec> and <DecList> by swapping the facotr in the left-recusive production and including an empty production. This results in the revised grammar segments below:

```
<DeclList>  → <Decl> <DecList>
            | ε

<VarDec>    → ID <VarDec>
            | ID ASSIGN <Exp>
            | ID COMMA <VarDec>
            | ε
```

Problem 9: Consider the CFG G = {NT = {E,T,F}, T = {a,b,+,*}, P, E } with the set of productions as follows:

(1) E → E + T
(2) E → T
(3) T → T F
(4) T → F
(5) F → F *
(6) F → a
(7) F → b

For the above grammar answer the following questions:

(a) Compute the FIRST and FOLLOW for all non-terminals in G.
(b) Consider the augmented grammar G' = { NT, T, { (0) E' -> E$ } + P, E' }. Compute the set of LR(0) items for G'.
(c) Compute the LR(0) parsing table for G'. If there are shift-reduce conflicts use the SLR parse table construction algorithm.
(d) Show the movements of the parser for the input w = "a+ab*$".
(e) Can this grammar be parsed by an LL (top-down) parsing algorithm? Justify.

Solution:

(a) We compute the FIRST and FOLLOW for the augmented grammar (0) E' → E$

FIRST(E)	= FIRST(T) = FIRST(F) = {a,b}
FOLLOW(E)	= {+,$}
FOLLOW(T)	= FIRST(F) + FOLLOW(E) = {a,b,+,$}
FOLLOW(F)	= {*,a,b,+,$}

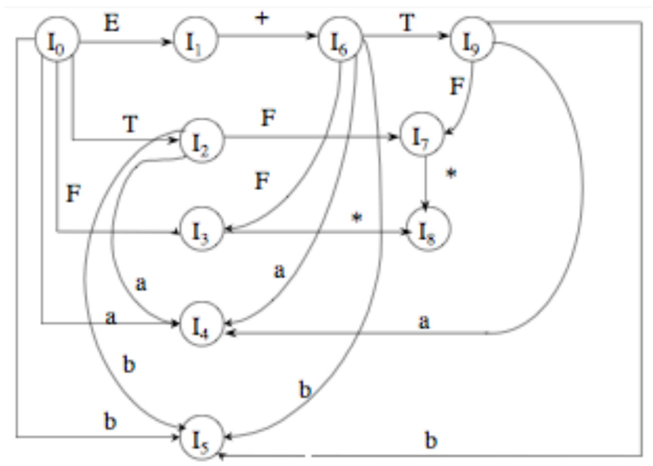
(b) Consider the augmented grammars E' → E$ we compute the LR(0) set of items.

```
I0 = closure({[E' → •E$]})
  = E' → •E$
    E → •E + T
    E → •T
    T → •T F
    T → •F
    F → •F *
    F → •a
    F → •b

I1 = goto (I0,E)
  = closure({[E' → E•$],[E → E• + T]})
  = E' → E• $
    E → E• + T

I2 = goto (I0,T)
   = closure({[E → T•],[T → T• F]})
   = E → E•
     T → T•F
     F → •F *
     F → •a
     F → •b

I3 = goto (I0, F)
    = closure({[T → F•],[F →F• *]})
    = T → F•
      F →F• *

I4 = goto (I0, a)
    = closure({[F → a•]})
    = F → a•

I5 = goto (I0, b)
    = closure({[F → b•]})
    = F → b•

I6 = goto (I1, +)
    = closure({[E → E+•T]})
    = E → E+•T
      T → •T F
      T → •F
      T → •F *
      F → •a
      F → •b

I3 = goto(I0 ,F)
   = closure({[T → F•], [F →F•*]})
```

I7 = goto (I2,F)
  = closure({[T → TF•],[F → F•*]})
  = T → T F•
    F → F•*
I8 = goto (I3,*)
  = closure({[F → F*•]})
  = F → F*•

I9= goto (I6, T)
  = closure({[E → E+T•], [E → T•F]})
  = E → E + T•
    E → T•F
    F → •F *
    F → •a
    F → •b

goto (I9,a) = I4
goto (I9,b) = I5
goto (I9,F) = I7



(c) We cannot construct an LR(0) parsing table because states I2, I3, I7 and I9 have shift-reduce conflicts as depicted in the table below (left). We use the SLR table-building algorithm, using the FOLLOW sets to eliminate the conflicts and build the SLR parsing table below (right). The CFG grammar in this exercise is thus an SLR grammar and not an LR(0) grammar.

state	Action					Goto		
	a	b	+	*	$	E	T	F
I0	shift I4	shift I5				goto I1	goto I2	goto I3
I1			shift I6		Accept			
I2	shift I4 reduce(2)	shift I5 reduce(2)	reduce(2)	reduce(2)	reduce(2)			goto I7
I3	reduce(4)	reduce(4)	reduce(4)	shift I8 reduce(4)	reduce(4)			
I4	reduce(6)	reduce(6)	reduce(6)	reduce(6)	reduce(6)			
I5	reduce(7)	reduce(7)	reduce(7)	reduce(7)	reduce(7)			
I6	shift I4	shift I5					goto I9	goto I3
I7	reduce(3)	reduce(3)	reduce(3)	shift I8 reduce(3)	reduce(3)			
I8	reduce(5)	reduce(5)	reduce(5)	reduce(5)	reduce(5)			
I9	shift I4 reduce(1)	shift I5 reduce(1)	reduce(1)	reduce(1)	reduce(1)			goto I7

state	Action					Goto		
	a	b	+	*	$	E	T	F
I0	shift I4	shift I5				goto I1	goto I2	goto I3
I1			shift I6		Accept			
I2	shift I4	shift I5	reduce(2)		reduce(2)			goto I7
I3	reduce(4)	reduce(4)	reduce(4)	shift I8	reduce(4)			
I4	reduce(6)	reduce(6)	reduce(6)	reduce(6)	reduce(6)			
I5	reduce(7)	reduce(7)	reduce(7)	reduce(7)	reduce(7)			
I6	shift I4	shift I5					goto I9	goto I3
I7	reduce(3)	reduce(3)	reduce(3)	shift I8	reduce(3)			
I8	reduce(5)	reduce(5)	reduce(5)	reduce(5)	reduce(5)			
I9	shift I4	shift I5	reduce(1)		reduce(1)			goto I7

(d) For example if input = “a+ab*$” the parsing is as depicted below where the parenthesis indicates a state symbol in the stack We also show the parser action and the corresponding grammar production in case of a reduce action.

$(I0)	shift I4	
$(I0)a(I4)	reduce(6)	F → a
$(I0)F(I3)	reduce(4)	T → F
$(I0)T(I2)	reduce(2)	E → T
$(I0)E(I1)	shift I6	
$(I0)E(I1)+(I6)	shift I4	
$(I0)E(I1)+(I6)a(I4)	reduce(6)	F → a
$(I0)E(I1)+(I6)F(I3)	reduce(4)	T → F
$(I0)E(I1)+(I6)T(I9)	shift I5	
$(I0)E(I1)+(I6)T(I9)b(I5)	reduce(7)	F → b
$(I0)E(I1)+(I6)T(I9)F(I7)	shift I8	
$(I0)E(I1)+(I6)T(I9)F(I7)*(I8)	reduce(5)	F → F*
$(I0)E(I1)+(I6)T(I9)F(I7)	reduce(3)	T → TF
$(I0)E(I1)+(I6)T(I9)	reduce(4)	E →E+T
$(I0)E(I1)	Accept	

(e) No, because the grammar is left-recursive.

Problem 10: LR Parsing

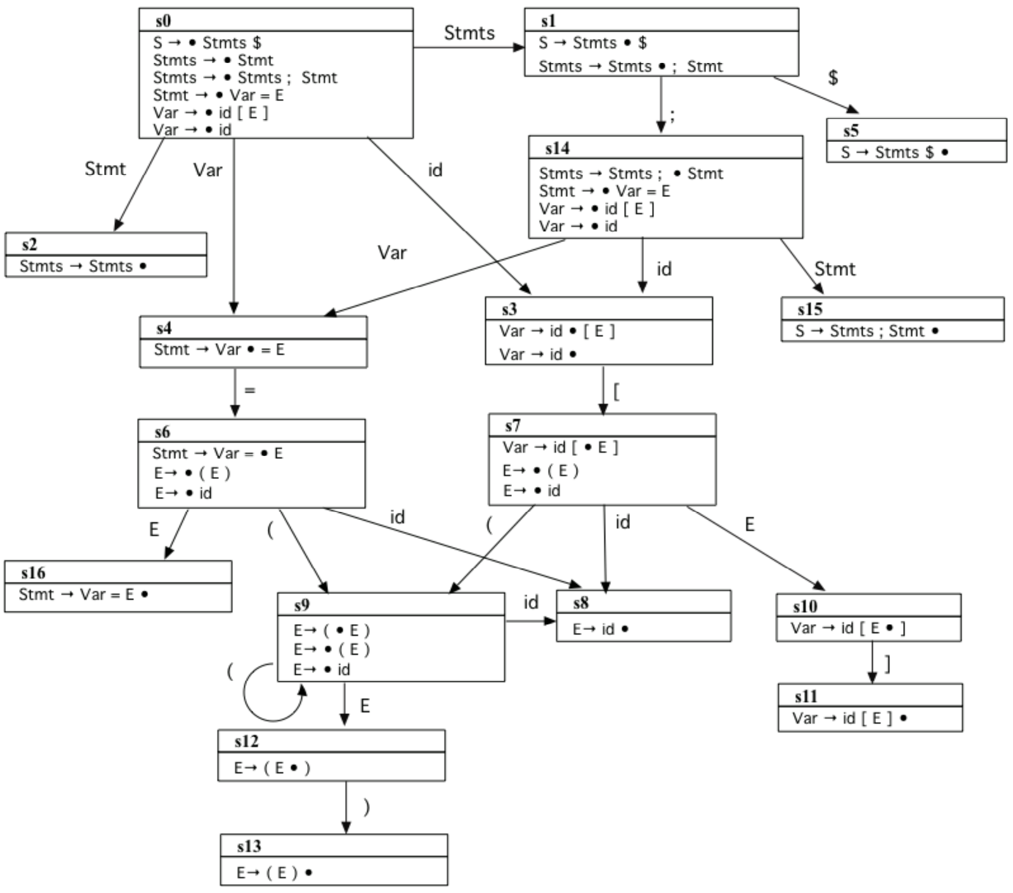
Given the grammar below already augmented production (0) answer the following questions:

```
(0) S       → Stmts $
(1) Stmts   → Stmt
(2) Stmts   → Stmts ; Stmt
(3) Stmt    → Var = E
(4) Var     → id [E ]
(5) Var     → id
(6) E       → id
(7) E       → ( E )
```

a) Construct the set of LR(0) items and the DFA capable of recognizing it.
b) Construct the LR(0) parsing table and determine if this grammar is LR(0). Justify.
c) Is the SLR(0) DFA for this grammar the same as the LR(0) DFA? Why?
d) Is this grammar SLR(0)? Justify by constructing its table.
e) Construct the set of LR(1) items and the DFA capable of recognizing it
f) Construct the LR(1) parsing table and determine if this grammar is LR(1). Justify.
g) How would you derive the LALR(1) parsing table this grammar? What is the difference between this table and the table found in a) above?

Solution:

a) Construct the set of LR(0) items and the DFA capable of recognizing it.
The figure below depicts the FA that recognizes the set of valid LR(0) items for this grammar.



b) Construct the LR(0) parsing table and determine if this grammar is LR(0). Justify.

Based on the DFA above we derive the LR parsing table below where we noted a shift/reduce conflict in state 3. In this state the presence of a '[' indicates that the parse can either reduce using the production 5 or shift by advancing to state s6. Note that by reducing it would then be left in a state possible s0 where the presence of the '[' would lead to an error. Clearly, this grammar is not suitable for the LR(0) parsing method.

	Terminals								Goto			
State	**id**	**;**	**=**	**[**	**]**	**(**	**)**	**$**	**Stmts**	**Stmt**	**E**	**Var**
0	s3								g1	g2		g4
1		s13						s5				
2	r(1)	r(1)	r(1)	r(1)	r(1)	r(1)	r(1)	r(1)				
3	r(5)	r(5)	r(5)	**s6/r(5)**	r(5)	r(5)	r(5)	r(5)				
4			s6									
5								acc				
6	s8					s9					g16	
7	s8					s9					g10	
8	r(6)	r(6)	r(6)	r(6)	r(6)	r(6)	r(6)	r(6)				
9	s8					s9					g12	
10					s11							
11	r(4)	r(4)	r(4)	r(4)	r(4)	r(4)	r(4)	r(4)				
12							s13					
13	r(7)	r(7)	r(7)	r(7)	r(7)	r(7)	r(7)	r(7)				
14	s3									g15		g4
15	r(2)	r(2)	r(2)	r(2)	r(2)	r(2)	r(2)	r(2)				
16	r(3)	r(3)	r(3)	r(3)	r(3)	r(3)	r(3)	r(3)				

c) Is the SLR(1) DFA for this grammar the same as the LR(0) DFA? Why?

The same. The states and transitions are the same as only the procedure to build the parse table is different. For this method of construction of the parsing table we include the production "reduce A $\rightarrow$ α" for all terminals "a" in FOLLOW(A). The table below is the resulting parse table using the SLR table construction algorithm, also known as SLR(1) although it uses the DFA constructed using the LR(0) items. For this specific grammar the FOLLOW set is as shown below:

FOLLOW(Stmts) = { $, ; }	FOLLOW(Stmt) = { $, ; }
FOLLOW(E) = { $, ; ,] ,) }	FOLLOW(Var) = { = }

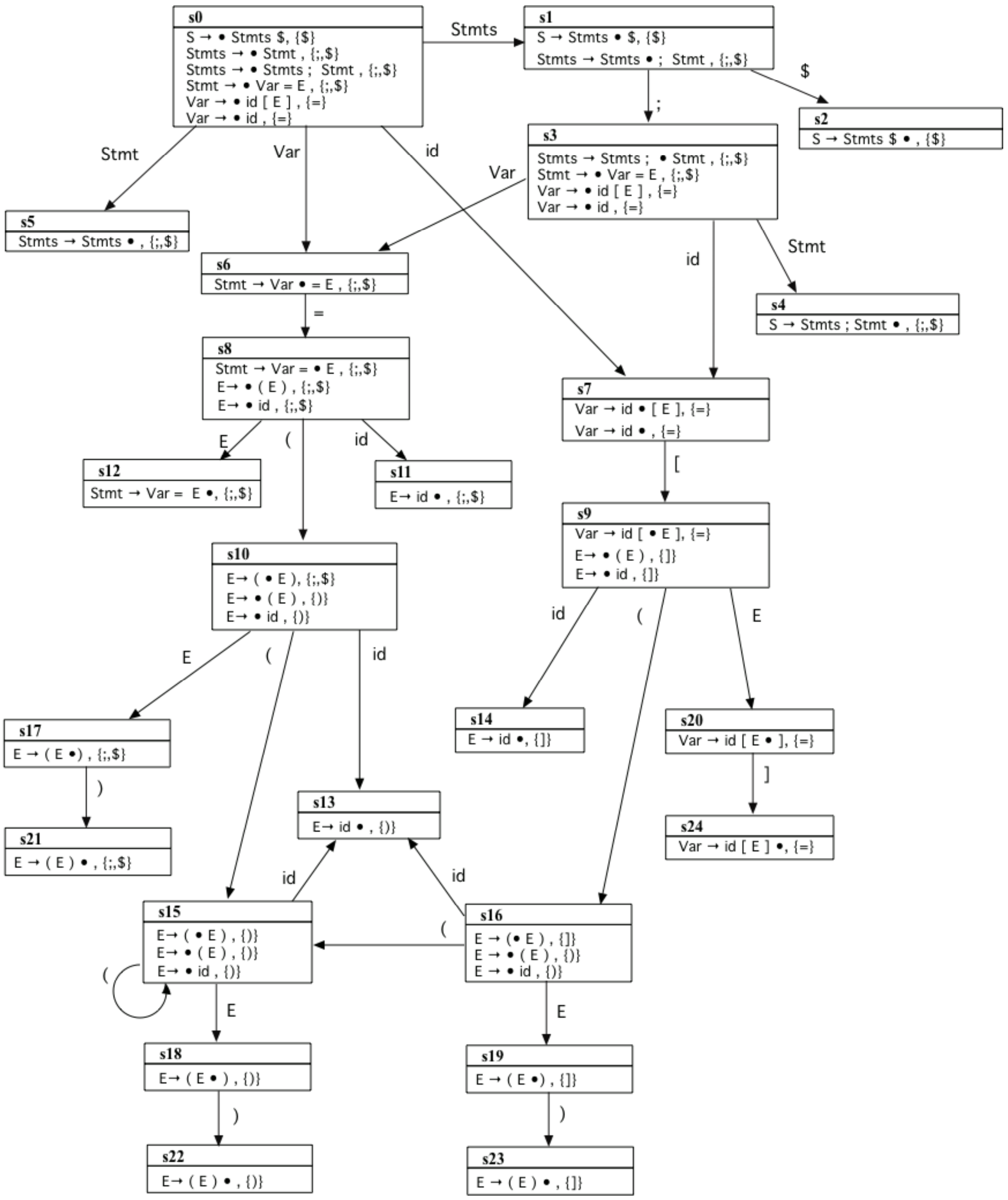
	Terminals								Goto			
State	**id**	**;**	**=**	**[**	**]**	**(**	**)**	**$**	**Stmts**	**Stmt**	**E**	**Var**
0	s3								g1	g2		g4
1		s13						s5				
2		r(1)						r(1)				
3			r(5)	s6								
4			s6									
5								acc				
6	s8					s9					g16	
7	s8					s9					g10	
8		r(6)			r(6)		r(6)	r(6)				
9	s8					s9					g12	
10					s11							
11			r(4)									
12							s13					
13		r(7)			r(7)		r(7)	r(7)				
14										g15		g4
15		r(2)						r(2)				
16		r(3)						r(3)				

Notice that because we have used the FOLLOW of Var to limit the use of the reduction action for this table we have eliminated the shit/reduce conflict in this grammar.

d) Is this grammar SLR(1)? Justify by constructing its table.

As can be seen in state 3 there is no longer a shift/reduce conflict. Essentially a single look-ahead symbol is enough to distinguish a single action to take in any context.

e) Construct the set of LR(1) items and the DFA capable of recognizing them.



As can be seen there number of states in this new DFA is much larger when compared to the DFA that recognizes the LR(0) sets of items.

f) Construct the LR(1) parsing table and determine if this grammar is LR(1). Justify.

	Terminals								**Goto**			
State	**id**	**;**	**=**	**[**	**]**	**(**	**)**	**$**	**Stmts**	**Stmt**	**E**	**Var**
0	s7								g1	g5		g6
1		s3						s2				
2								acc				
3	s7									g4		g6
4		r(2)						r(2)				
5		r(1)						r(1)				
6			s8									
7			r(5)	s9								
8	s11					s10					g12	
9	s14					s16					g20	
10	s13					s15					g17	
11		r(6)						r(6)				
12		r(3)						r(3)				
13							r(6)					
14					r(6)							
15	s13					s15					g18	
16	s13					s15					g19	
17							s21					
18							s22					
19							s23					
20					s24							
21		r(7)					r(7)					
22							r(7)					
23					r(7)							
24			r(4)									

Clearly, and as with the SLR(1) table construction method there are no conflicts in this parse table and the grammar is therefore LR(1).

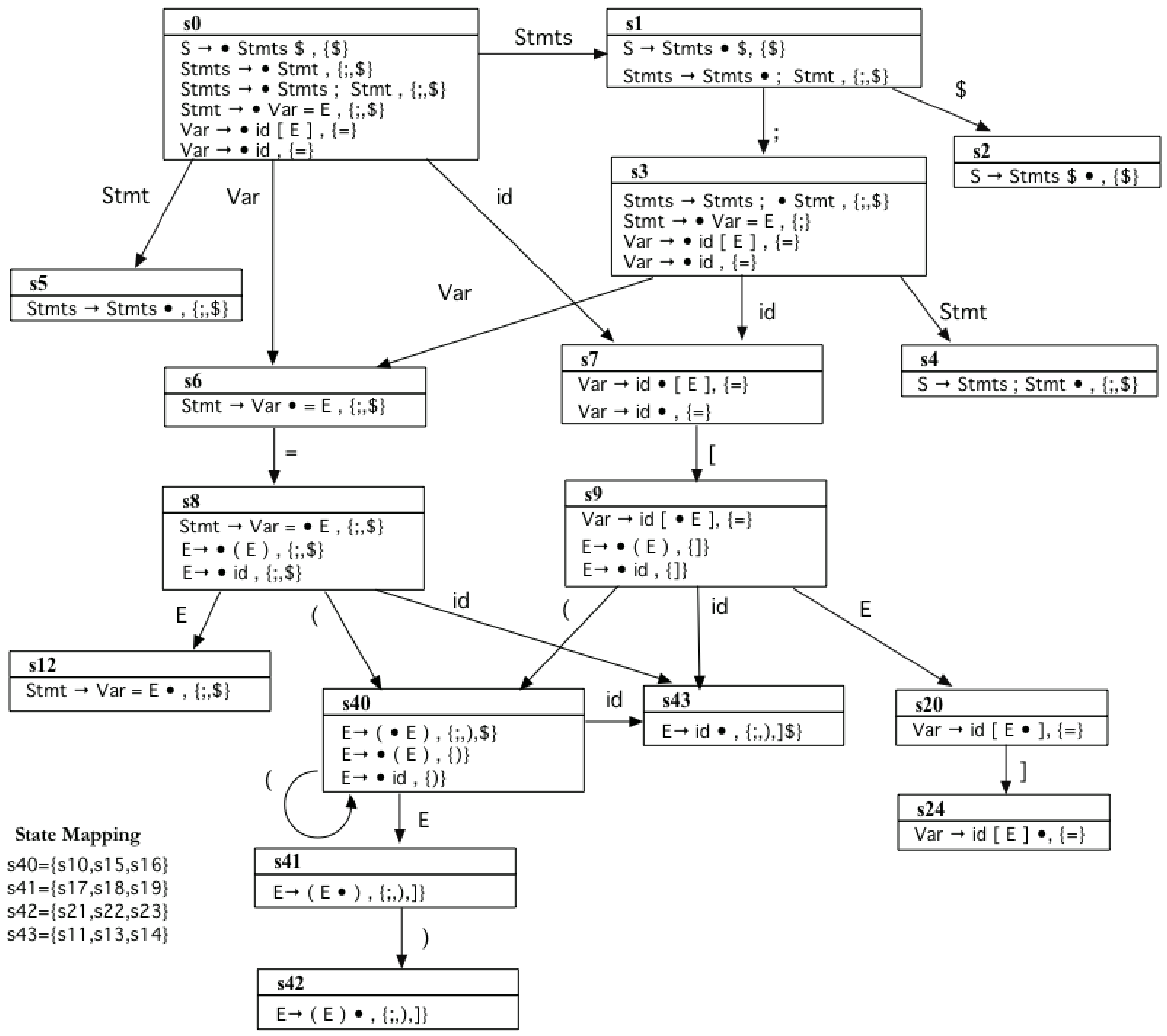
g) How would you derive the LALR(1) parsing table this grammar? What is the difference between this table and the table found in a) above?

There are many states with very similar core items that differ only on the look-ahead and can thus be merged as suggested by the procedure to construct LALR parsing tables. There are many states with identical core items thus differing only in the look-ahead and can thus be merged as suggested by the procedure to construct LALR parsing tables. For instance states {s10, s15, s16} could be merged into a single state named s40. The same is true for states in the sets s41={s17, s18, s19}, s42={s21, s22, s23} and s43={s11, 13, s14} thus substantially reducing the number of states as it will be seen in the next point in this exercise.

The table below reflects these merge operations resulting in a much smaller table which is LALR as there are no conflicts due to the merging of states with the same core items.

	Terminals								Goto			
State	**id**	**;**	**=**	**[**	**]**	**(**	**)**	**$**	**Stmts**	**Stmt**	**E**	**Var**
0	s7								g1	g5		g6
1		s3						s2				
2								acc				
3	s7									g4		g6
4		r(2)										
5		r(1)						r(1)				
6			s8									
7			r(5)	s9								
8	s43					s40					g12	
9	s43					s40					g20	
40	s43					s40					g41	
41							s42					
42		r(7)			r(7)		r(7)					
43		r(6)			r(6)		r(6)	r(6)				

After this state simplification we get the DFA below. This DFA is identical to the first DFA found using the LR(0) items but the additional information on the look-ahead token allows for a better table parsing construction method that does not have shift/reduce conflicts.



Problem 11: Consider the following Context-Free Grammar G = ({S,A,B},S,{a,b},P) where P is

(1) S → Aa
(2) S → bAc
(3) S → dc
(4) S → bda
(5) A → d

Show that this grammar is LALR(1) but not SLR(1). To show this you need to construct the set of LR(0) items and see that there is at least one multiply defined entry in the SLR table. Then compute the set of LR(1) items and show that the grammar is indeed LALR(1). Do not forget to use the augmented grammar with the additional production { S '→ S $ }.

Solution:

To begin with, we compute the FIRST and FOLLOW sets for S and A, as FIRST(S) = {b,d} and FIRST(A) ={d} and FOLLOW(A) = {a,c}, FOLLOW(S) = {$} used in computing the SLR table.

We now compute the set of LR(0) items

```
I0 = closure({S '→•S $}) =
        S '→ • S $
        S  → • Aa
        S  → • bAc
        S  → • bda
        S  → • dc
        A  → • d

I1 = goto(I0,S) = closure({S '→ S •$}) =
        S '→ S •$

I2 = goto(I0,A) = closure({S → A•a}) =
        S → A•a

I3 = goto(I0,b) = closure({S → b•Ac, S → b•da }) =
        S → b•Ac
        S → b•da
        A → •d

I4 = goto(I0,d) = closure({S → d•c, A → d• }) =
        S  → d•c
        A → d•

I5 = goto(I2,a) = closure({S → Aa•}) =
        S → Aa•

I6 = goto(I3,a) = closure({S → bA•c}) =
        S → bA•c

I7 = goto(I3,d) = closure({S → bd•a, A→ d•}) =
        S → bd•a
        A→ d•

I8 = goto(I4,c) = closure({S → dc•}) =
        S → dc•

I9 = goto(I6,c) = closure({S → bAc•}) =
        S → bAc•

I10 = goto(I7,a) = closure({S → bda•}) =
        S → bda•
```

The parsing table for this grammar would have a section corresponding to states I4 and I7 with conflicts. In states I4 on the terminal c the item S → d•c would prompt a shift on c but since FOLLOW(A) = {a,c} the item A → d• would create a reduce action on that same entry, thus leading to a shift/reduce conflicts. A similar situation arises for state I7 but this time for the terminal a. As such this grammar is not SLR(1).

To show that this grammar is LALR(1) we construct its LALR(1) parsing table. We need to compute first the LR(1) sets of items.

I0 = closure({S '→•S $, $}) =
S '→ • S $, $
S → • Aa, $
S → • bAc, $
S → • bda, $
S → • dc, $
A → • d, a

I1 = goto(I0,S) = closure({S '→ S •$, $}) =
S '→ S •$, $

I2 = goto(I0,A) = closure({S → A•a, $}) =
S → A•a, $

I3 = goto(I0,b) = closure({[S → b•Ac, $], [S → b•da,$]}) =
S → b•Ac, $
S → b•da, $
A → •d, $/c

I4 = goto(I0,d) = closure({[S → d•c ,$],[A → d•, a]}) =
S → d•c,$
A → d•, a

I5 = goto(I2,a) = closure({S → Aa•,$}) =
S → Aa•,$

I6 = goto(I3,a) = closure({S → bA•c,$}) =
S → bA•c,$

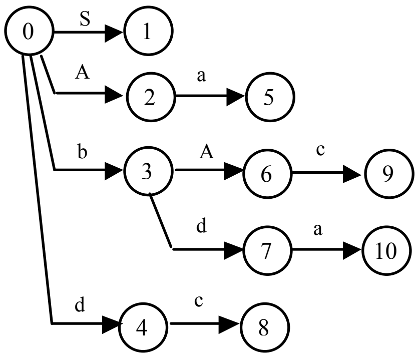
I7 = goto(I3,d) = closure({S → bd•a,$, A→ d•,$c}) =
S → bd•a,$
A→ d•,$/c

I8 = goto(I4,c) = closure({S → dc•,$}) =
S → dc•,$

I9 = goto(I6,c) = closure({S → bAc•,$}) =
S → bAc•,$

I10 = goto(I7,a) = closure({S → bda•,$}) =
S → bda•,$

In this case, and since we do not have two sets with identical core items, the LALR(1) and LR(1) parsing tables are identical. The DFA build from the set of items and the table is shown below.


$$\begin{tikzcd}
0 \arrow[r, "S"] \arrow[dr, "A"] \arrow[ddr, "b"] \arrow[ddddr, "d"] & 1 & & \\
& 2 \arrow[r, "a"] & 5 & \\
& 3 \arrow[r, "A"] \arrow[dr, "d"] & 6 \arrow[r, "c"] & 9 \\
& & 7 \arrow[r, "a"] & 10 \\
& 4 \arrow[r, "c"] & 8 &
\end{tikzcd}$$

State	Action					Goto	
	a	b	c	D	$	S	A
0		shift 3		shift 4		1	2
1					accept		
2	shift 5						
3				shift 7			6
4	reduce (5)		shift 8				
5					reduce (1)		
6			shift 9				
7	shift 10		reduce (5)		reduce (5)		
8					reduce (3)		
9					reduce (2)		
10					reduce (4)		

This is an LALR(1) parsing table without any conflicts, thus the grammar is LALR(1).

Problem 12: Given the following CFG grammar G = ({SL}, S, {a, "(",")",","), P) with P:

S → (L) I a
L → L, S I S

Answer the following questions:

a) Is this grammar suitable to be parsed using the recursive descendent parsing method? Justify and modify the grammar if needed.
b) Compute the FIRST and FOLLOW set of non-terminal symbols of the grammar resulting from your answer in a)
c) Construct the corresponding parsing table using the predictive parsing LL method.
d) Show the stack contents, the input and the rules used during parsing for the input string w = (a,a)

Solution:

a) No because it is left-recursive. You can expand L using a production with L as the left-most symbol without consuming any of the input terminal symbols. To eliminate this left recursion we add another non-terminal symbol, L' and productions as follows:

S → (L) I a
L → S L'
L' → , S L' I ε

b) FIRST(S) = { (, a }, FIRST(L) = { (, a } and FIRST(L') = { , , ε}
FOLLOW(L) = {) }, FOLLOW(S) = { , ,) , $ }, FOLLOW(L') = {) }

c) The parsing table is as shown below:

	(	)	a	,	$
S	S→(L)		S → a		
L	L→S L'		L→ S L'		
L'		L'→ε		L'→ , S L'	

d) The stack and input are as shown below using the predictive, table-driven parsing algorithm:

STACK	INPUT	RULE/OUTPUT
$S	(a,a)$	
$) L (	(a,a)$	S → (L)
$) L	a,a)$	
$) L' S	a,a)$	L → S L'
$) L' a	a,a)$	S → a
$) L'	,a)$	
$) L' S ,	,a)$	L' → , S L'
$) L' S	a)$	
$) L' a	a)$	S → a
$) L'	)$	
$)	)$	S → ε
$	$	

Problem 13: In class we saw an algorithm used to eliminate left-recursion from a grammar G. In this exercise you are going to develop a similar algorithm to eliminate ε-productions, i.e., productions of the form A → ε. Note that if ε is in L(G) you need to retain a least one ε-production in your grammar as otherwise you would be changing L(G). Try your algorithm using the grammar G = {S, {S},{a,b},{S→aSbS, S→bSa, S→ ε}}.

Solution:

Rename all the non-terminal grammar symbols, A1, A2, …, Ak, such that an ordering is created (assign A1 = S, the start symbol).

(1) First identify all non-terminal symbols, Ai, that directly or indirectly produce the empty-string (i.e. epsilon production)

Use the following 'painting' algorithm:

1. For all non-terminals that have an epsilon production, paint them blue.
2. For each non-blue non-terminal symbol Aj, if Aj→W1W2…Wn is a production where Wi is a non-terminal symbol, and Wi is blue for i=1,…,n, then paint Aj blue.
3. Repeat step 2, until no new non-terminal symbol is painted blue.

(2) Now for each production of the form A → X1 X2 … Xn, add A→W1 W2 … Wn such that:

(i) if Xi is not painted blue, Wi = Xi
(ii) if Xi is painted blue, Wi is either Xi or empty
(iii) not all of Wi are empty.

Finally remove all A→ epsilon productions from the grammar.

If S → ε, augment the grammar by adding a new start symbol S' and the productions:

S' → S
S' → ε

Applying this algorithm to the grammar in the question yields the equivalent grammar below:

S'→ S | ε
S → aSb | bSa | abS | ab | ba

Problem 14: Given the grammar G = {S,{S,A,B},{a,b,c,d},P} with set of productions P below compute;

a. LR(1) sets of items
b. The corresponding parsing table for the corresponding shift-reduce parse engine
c. Show the actions of the parsing engine as well as the contents of the symbol and state stacks for the input string w ="bda$".
d. Is this grammar LALR(1)? Justify.

(1) S → Aa
(2) | bAc
(3) | Bc
(4) | bBa
(5) A → d

(6) B → d

Do not forget to augment the grammar with the production S' → S$

Solution:

(a) LR(1) set of items

I0 = closure{[S' → .S, $]}
S'→ .S, $
S → .Aa, $
S → .bAc, $
S → .Bc, $
S →.bBa, $
A →.d, a
B →.d, c

I1 = goto(I0, S)
S'->S., $

I2 = goto(I0, A)
S → A.a, $

I3 = goto(I0, B)
S → B.c, $

I4 = goto(I0, b)
S → b.Ac, $
S → b.Ba, $
A →. d, c
B → .d, a

I5 = goto(I0, d)
A→ d., a
B → d., c

I6 = goto(I2, a)
S → Aa., $

I7 = goto(I3, c)
S → Bc., $

I8 = goto(I4, A)
S → bA.c, $

I9 = goto(I8, c)
S → bAc., $

I10 = goto(I4, B)
S → bB.a, $

I11 = goto(I10, a)
S → bBa., $

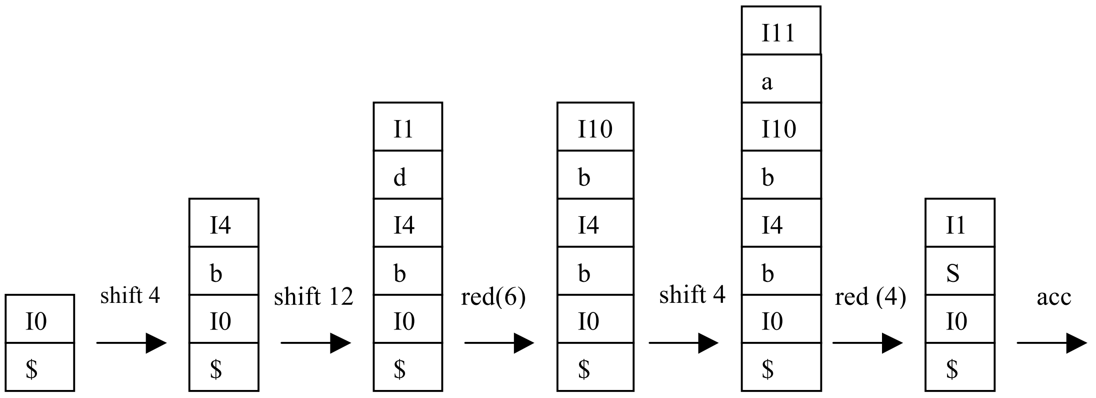
I12 = goto(I4, d)
A→ d., c
B → d., a

$$\begin{tikzcd}
I0 \arrow[r, "S"] \arrow[dr, "A"] \arrow[ddr, "B"] \arrow[dddr, "d"] \arrow[ddddr, "b"] & I1 & & \\
& I2 \arrow[r, "a"] & I6 & \\
& I3 \arrow[r, "c"] & I7 & \\
& I5 & & \\
& I4 \arrow[r, "A"] \arrow[dr, "B"] \arrow[ddr, "d"] & I8 \arrow[r, "c"] & I9 \\
& & I10 \arrow[r, "a"] & I11 \\
& & I12 &
\end{tikzcd}$$

(b) Parsing Table:

Start	Action					Goto		
	a	b	c	d	$	S	A	B
I0		shift I4		shift I5		goto I1	goto I2	goto I3
I1					Accept			
I2	shift I6							
I3			shift I7					
I4				shift I12			goto I8	goto I10
I5	reduce (5)		reduce (6)					
I6					reduce (1)			
I7					reduce (3)			
I8			shift I9					
I9					reduce (2)			
I10	shift I11							
I11					reduce (4)			
I12	reduce (6)		reduce (5)					

(c) For input string w=”bda$”, we follow the information in the table, starting from the bottom of the stack with the EOF $ symbol and state I0.



(d) In this particular case the only states that have common core items are states I5 and I12. If we merge them, however, we would get a state I512 with four items A→ d . , {a,c} and B → d . , {a,c}. This would mean that the corresponding parsing table would now have two reduce/reduce conflicts on the rules (5) and (6). This means that this particular grammar is not LALR(1).