CS143 Handout #16
Autumn 2000

Midterm practice solutions

1)

- **F** The language {(), (()), ((()))} is not parsable by any LL(1) parser.
- **T** Given an SLR(1) parser and an LALR(1) parser for the same grammar, both have the same number of states.
- **F** A grammar is ambiguous if there are several derivations for the same string.
- **F** NFAs can recognize more languages than DFA due to the added expressive power of non-determinism and ε-transitions.
- **T** A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop.
- **T** Empty entries in the Goto table of an LR(1) parser are not useful for detecting parse errors.
- **T** An SLR(1) parser operates in O(n) time, where n is the length of the input string.
- **F** Every LL(1) grammar is SLR(1).
- **T** Even if the language of a CFG is regular, the grammar may not be LL(1).
- **F** No right-recursive grammar can be LR(1).
- **T** An SLR(1) parser will never shift an erroneous symbol onto the parse stack, i.e. only symbols that can possibly be reduced will ever be shifted.
- **F** A table-driven top-down predictive parser is capable of parsing more grammars than a recursive-descent non-backtracking parser.
- **F** Consider a grammar with no e-productions and no single productions (i.e. no productions with a single symbol on the right-hand side). For an input of n tokens, a bottom-up parser can make a maximum of 2*n reductions.

2)

a) The minus is not always the unary minus! Consider parsing the input `8-4`. If the lexer insists on grouping the minus with the 4, then the parser will not be able to recognize this as a binary subtraction expression. The lexer does not have the context to determine whether the minus is a part of the integer token or an operator. It should always treat it a separate token and pass it to the parser who can decide how to interpret it relative to the surrounding context.

b) You cannot write a CFG to express this constraint because it requires context-sensitivity. Thus you would not be able to check this requirement using the parsing techniques we have studied. Ensuring the names matched would be a job for the semantic analyzer, the phase following syntax analysis.

c) One straightforward way to construct such a grammar is to establish a non-terminal for each qualifier subset (i.e. non-terminal `AB` is subset containing options `a` and `b`, `ABC` is for a, `b`, and c, and so on) and for the start symbol to expand to any subset non-terminal. Each subset non-terminal expands to all the permutations of the subset. The productions are constructed by choosing one of the options from the subset to be the first, followed by the non-terminal that expands the remaining subset. If n is the number of type qualifiers, there will be 2^n such non-terminals in the grammar and each will have m productions, where m is the size of the subset. Adding a new qualifier means doubling the number of non-terminals (need both subsets with and without new qualifier) and increasing the number of productions in the new non-terminals by one (because of the additional option).As you can imagine, this quickly becomes unwieldy. Imposing an order on the options makes it much simpler, for each option you need only one non-terminal, which expands either to the option or null. The start symbol just expands to the non-terminals in the specified order. Adding a new qualifier to this version requires adding an additional non-terminal with two productions and inserting that non-terminal in its proper place in the start symbol. Languages like C and Java that have a interchangable lists of type qualifiers don't try to use the parser for this, the grammar is written to allow any number of the options to be specified and the non-repetition, non-clash constraint is handled as part of semantic analysis.

d) An LL(2) would be able to parse the grammar below and LL(1) would not. With two symbols of lookahead, the parser would see both the a and b, allowing it to distinguish which of the two productions to predict. With only one symbol of lookahead, either is possible and the LL(1) parser would not be able to predict which production to expand.

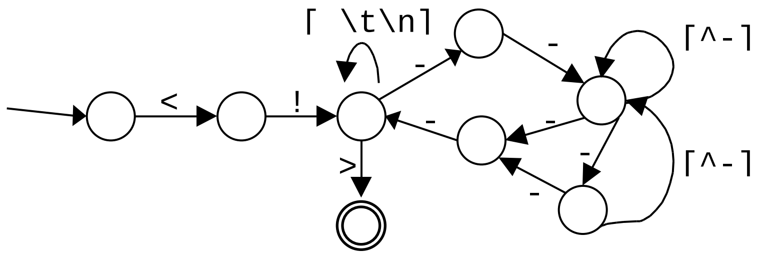
S ← aa | ab

e) (See problem set 2 solutions)

3) a)
```
Cons ([^aeiou])
%%
{Cons}*(a{Cons}*)*(e{Cons}*)*(i{Cons}*)*(o{Cons}*)*(u{Cons}*)*
```

b)
```
xdanjagbeont|iutpu|acene|au
```

4) a)



b)
```
White ([^ \t\n]*)
Comm (dd([^d]|d[^d])*d?dd)
%%
<!({White}{Comm}*{White})*>
```

5) a)

F	←id(L)
L	←PL' | ∂
L'	←, PL' | ∂
P	←int | idP'
P'	←(L)| ∂

b)

First(F) = { id }
First(L) = { int id ∂ }
First(L') = { , ∂ }
First(P) = { int id }
First(P') = { (∂ }
Follow(F) = { $ }
Folllow(L) = Follow(L') = {) }
Follow(P) = Follow(P') = { ,) }

c)

Stack	int	id	,	(	)	$
F		F ←id(L)				
L	L ←PL'	L ←PL'			L ←ϴ	
L'			L' ←,PL'		L' ←ϴ	
P	P ←int	P ←idP'				
P'			P' ←ϴ	P' ←(L)	P' ←ϴ	

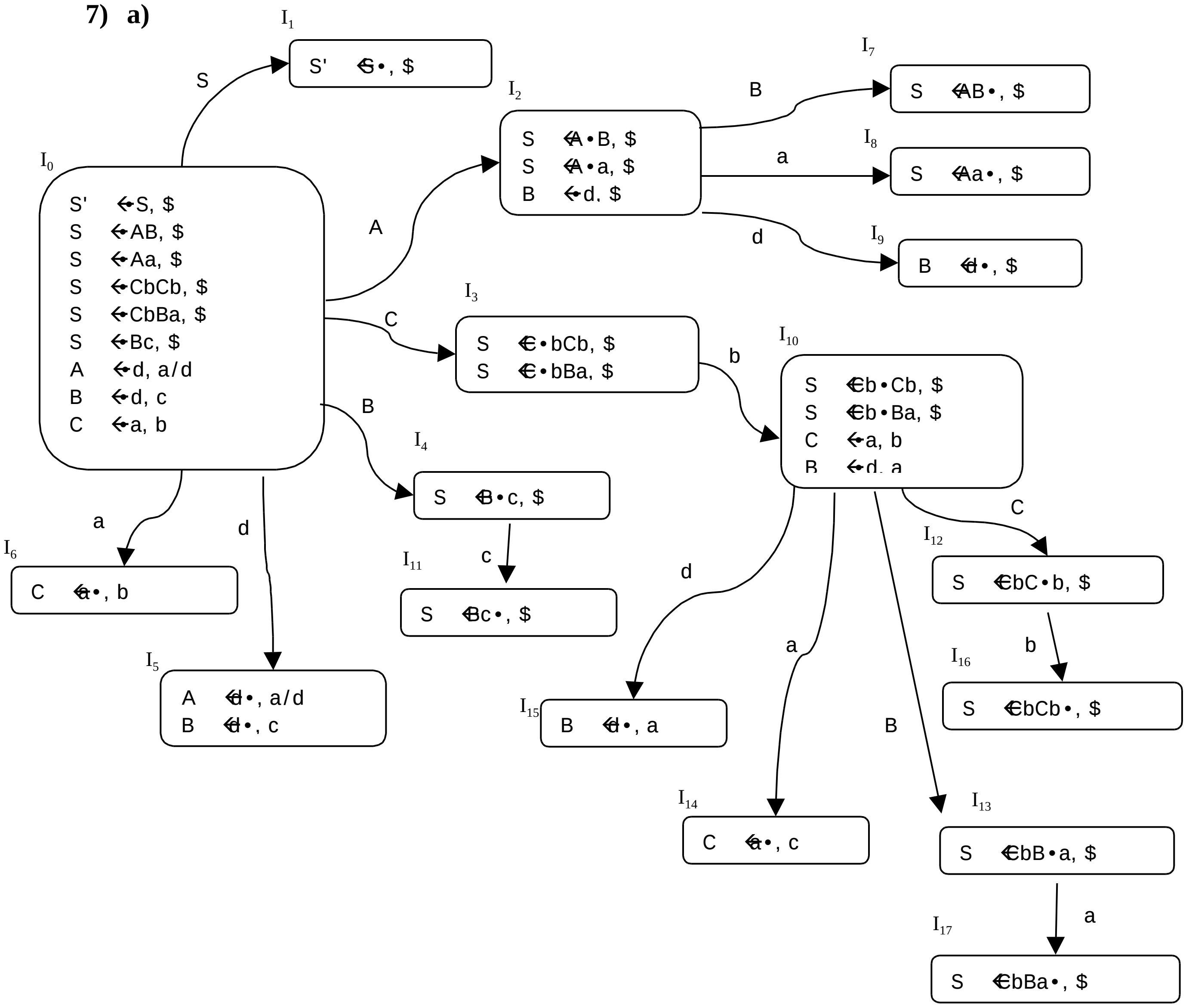
d)

PARSE STACK	REMAINING INPUT	PARSER ACTION
F$	Binky(4, Winky())$	Predict F ←id(L)
id(L)$	Binky(4, Winky())$	Match id
(L)$	(4, Winky())$	Match (
L)$	4, Winky())$	Predict L ←PL'
PL')$	4, Winky())$	Predict P ←int
intL')$	4, Winky())$	Match int
L')$	, Winky())$	Predict L' ←,PL'
,PL')$	, Winky())$	Match ,
PL')$	Winky())$	Predict P->idP'
idPL')$	Winky())$	Match id
P'L')$	())$	Predict P' ←(L)
(L)L')$	())$	Match (
L)L')$	))$	Predict L ←ϴ
)L')$	))$	Match)
L')$	)$	Predict L' ←ϴ
)$	)$	Match)
$	$	Match $, success!

6) Given the first and follow sets below, there will be a conflict when A is on top of the stack and the next input is b or c. The parser could predict $A \rightarrow \epsilon$ since b and c are members of Follow(A) or predict $A \rightarrow bAb$ (for b) or $A \rightarrow cSc$ (for c). Note there is no conflict in the table for the S row on input b or c, the table predicts $S \rightarrow A$.

First(S) = { a b c ∂ }
First(A) = { b c ∂ }
Follow(S) = { b c $ }
Follow(A) = { b c $ }

7) a)



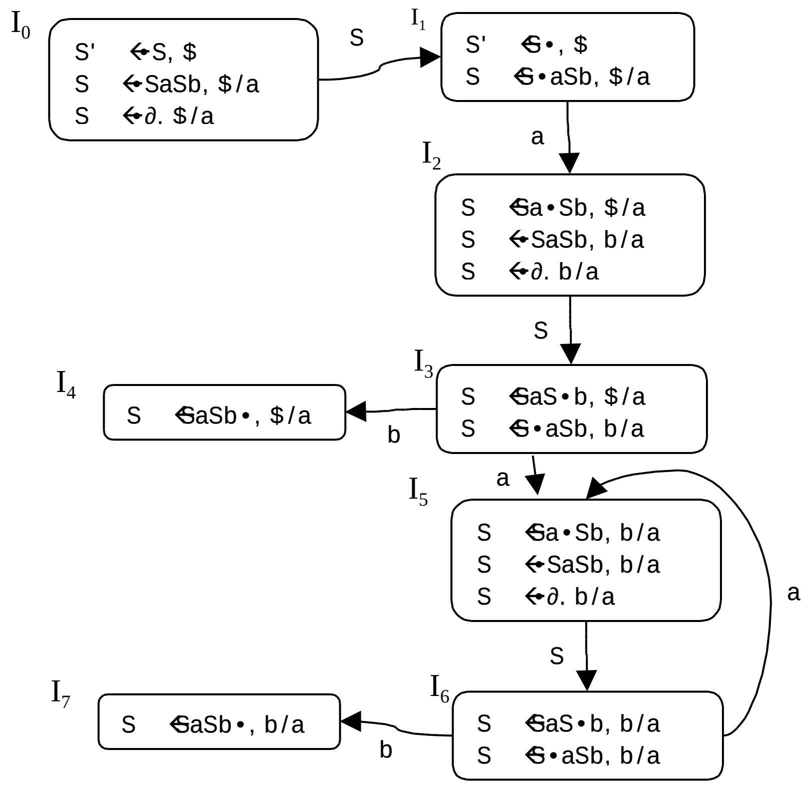
b)

State	Action					Goto			
	a	b	c	d	$	S	A	B	C
0	s6			s5		1	2	4	3
1					acc			7	
2	s8			s9					
3		s10							
4			s11						
5	r7		r6	r7					
6		r8							
7					r1				
8					r2				
9					r6				
10	s14			s15				13	12
11					r5				
12		s16							
13	s17								
14			r8						
15	r6								
16					r3				
17					r4				

c) Yes. There are no conflicts in the LR(1) parse table.
d) No. In state 5, we have d on the stack which can reduce to A or B. The follow sets for A and B are not disjoint, we have a reduce/reduce conflict on next input a.
e) Yes. Merging states 6 and 14 and states 9 and 15 would not add reduce/reduce conflicts.

8) **a)** The grammar is SLR(1). (See configurating sets below, no conflicts even in follow sets)
b) LALR merges states 2 and 5, states 3 and 6, and states 4 and 7, thus the table has three fewer rows than the LR.
c) Using Follow(S) adds a few extra reductions in states 0 and 25, so the tables are different.

There is no "quick" way to determine if a grammar is LR/SLR/LALR, you build the configurating sets and look for conflicts. There are some short-cuts (an ambiguous grammar can't be LR(1), if the follow sets for all non-terminals are disjoint, it is probably SLR(1), etc.) but they don't address the general problem. For the grammar above, there is no possible of any reduce/reduce conflicts since there is at most one possible reduction in each state. State 1 is the only state that has both a shift and reduce and there is no overlap for the input symbols, so there is also no chance of a shift/reduce conflict.



9) That would be Montpelier, Vermont.