Discussion5 Answer

1. **Shift-Reduce Parsing the Lambda Calculus.**

   We'll look again at the lambda calculus grammar:

```
1. expr : var
2.      | ‘(’ ‘λ’ var ‘.’ expr ‘)’
3.      | ‘(’ expr expr ‘)’ ;
4. var  : ID ;
```

   (a) Is this grammar LL(1)?

   (b) We'll now use the following LR(1) parsing table to parse some strings with this grammar.

	(	)	.	ID	λ	\$	var	expr
0	s1			s2			s4	s3
1	s1			s2	s5		s4	s6
2	r4	r4	r4	r4	r4	r4		
3						s7		
4	r1	r1	r1	r1	r1	r1		
5				s2			s8	
6	s1			s2			s4	s9
7	ACC	ACC	ACC	ACC	ACC	ACC		
8			s10					
9		s11						
10	s1			s2			s4	s12
11	r3	r3	r3	r3	r3	r3		
12		s13						
13	r2	r2	r2	r2	r2	r2		

   Here's how we use this parsing table:

   - Maintain a stack of states. Initialize it with state 0.
   - Use the state on top of the parse stack and the lookahead symbol to find the corresponding action in the parse table.
     - If the action is a shift to a new state, push the new state onto the stack and advance the input.

- If the action is a reduce that reduces k symbols, pop k symbols off the stack. If we reduced to a rule A, consult the "A" entry in the state now at the top of the stack.

- If the action is ACC, then we have succeeded.

Use the parsing table above to parse the following:

i. (xx)
ii. $(\lambda x.x)$
iii. $(x(\lambda x.x))$

(c) Is this grammar LR(0)?

2. **Altering the Lambda Calculus.**

Suppose we want to add an optional extension that allows raising a `var` to a power. We define the grammar as

```
expr : var
     | var '^' NUM
     | '(' 'λ' var '.' expr ')'
     | '(' expr expr ')' ;
var  : ID ;
```

(a) Is this grammar LR(0)?

(b) Which state in the parsing table would we need to modify to parse this grammar?

3. **Stack in Shift-Reduce Parsing.**

Suppose it is given that shift-reduce parsing is equivalent to finding the rightmost derivation in reverse. Prove that during shift-reduce parsing, we can only reduce the topmost items in the stack (i.e. we don't need to worry about reducing something in the middle; hence the usage of a stack is justified).